discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code

Sebastian Eschweiler*†, Khaled Yakdan*†, Elmar Gerhards-Padilla†,
*University of Bonn, Germany
{yakdan, eschweil}@cs.uni-bonn.de
†Fraunhofer FKIE, Germany
elmar.gerhards-padilla@fkie.fraunhofer.de

***Abstract*—The identification of security-critical vulnerabilities is a key for protecting computer systems. Being able to perform this process at the binary level is very important given that many software projects are closed-source. Even if the source code is available, compilation may create a mismatch between the source code and the binary code that is executed by the processor, causing analyses that are performed on source code to fail at detecting certain bugs and thus potential vulnerabilities. Existing approaches to find bugs in binary code 1) use dynamic analysis, which is difficult for firmware; 2) handle only a single architecture; or 3) use semantic similarity, which is very slow when analyzing large code bases.**

In this paper, we present a new approach to efficiently search for similar functions in binary code. We use this method to identify known bugs in binaries as follows: starting with a vulnerable binary function, we identify similar functions in other binaries across different compilers, optimization levels, operating systems, and CPU architectures. The main idea is to compute similarity between functions based on the structure of the corresponding control flow graphs. To minimize this costly computation, we employ an efficient pre-filter based on numeric features to quickly identify a small set of candidate functions. This allows us to efficiently search for similar functions in large code bases.

We have designed and implemented a prototype of our approach, called discovRE, that supports four instruction set architectures (x86, x64, ARM, MIPS). We show that discovRE is four orders of magnitude faster than the state-of-the-art academic approach for cross-architecture bug search in binaries. We also show that we can identify *Heartbleed* and *POODLE* vulnerabilities in an Android system image that contains over 130,000 native ARM functions in about 80 milliseconds.

I. Introduction

One key problem in computer security is the identification of bugs and security-critical vulnerabilities in software. Despite intensive efforts by the research community and the industry, vulnerabilities continue to emerge regularly in popular software projects. Unfortunately, a single flaw in program code, such as the failure to check buffer boundaries or sanitize input data, can render the whole program vulnerable, which can have a huge security impact given that popular software programs are used by millions of people on a daily basis. Prominent recent examples of these cases include the *Heartbleed* vulnerability in the cryptographic library OpenSSL [5], the "Shellshock" vulnerability in GNU Bash [8], and the *POODLE* vulnerability in the SSLv3 protocol [7]. Given the evolving nature of programs and the increasing complexity, efficient and accurate approaches to identify vulnerabilities in large code bases are needed.

There has been an extensive research on identifying vulnerabilities at the source code level. Security research has focused on finding specific types of vulnerabilities, such as buffer overflows [66], integer-based vulnerabilities [17, 59], insufficient validation of input data [37], or type-confusion vulnerabilities [42]. A similar line of research to our work is identifying bugs in source code by searching for code fragments similar to an already known venerability [27, 34, 35, 38, 63]. Unfortunately, performing bug search at the source level only is not sufficient for two reasons: first, many prominent and popular software projects such as MS Office, Skype, and Adobe Flash Player are closed-source, and are thus only available in the binary form. Second, compilation optimizations may change some code properties, creating a mismatch between the source code of a program and the compiled binary code that is executed on the processor [15]. For example, zeroing a buffer containing sensitive information before freeing it may be marked as useless code and thus removed by the compiler since the written values are never used at later stages. As a result, performing analysis on the source code may fail to detect certain vulnerabilities. This illustrates the importance of being able to perform bug search at the binary level.

Bug search at the binary level is very challenging. Due to the NP-complete nature of the compiler optimization problem [12], even recompiling the same source code with the same compiler and optimization options can potentially alter the resulting binary. In many cases the same source code is compiled using different toolchains, i.e., compilers or optimization levels. This can heavily alter the generated binary code, making it extremely difficult to identify the binary code fragments that stemmed from the same source code fragment. This is even further complicated by the fact the same source code can be cross-compiled for different CPU architectures, which results in binaries that differ, among others, in instruction sets, register names, and function calling conventions.

Previous work to identify bugs in binary code either relied on dynamic analysis [e.g., 23], supported a single architecture [e.g., 26], or used semantic similarity [e.g., 53, 44]. Dynamic analysis relies on architecture-specific tools to execute

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23185

or emulate binaries. As a result, extending these approaches to support other architectures would be tedious. Moreover, code coverage is a fundamental shortcoming for any dynamic approach, limiting the amount of code that can be searched for bugs. Approaches based on semantic similarity provide the most accurate results. However, SAT/SMT solvers which are often used to measure semantic similarity are computationally expensive. For example, the most recent and advanced method to perform cross-architecture bug search in binary code needs on average 286.1 minutes for each 10,000 basic blocks to prepare a binary for search. This means that it would need about one month to prepare a stock Android image that contains about 1.4 million basic blocks. This motivated us develop a new and efficient bug search method that can be used with realistically large code bases.

In this paper, we aim at improving the state of the art by presenting a new approach for robust and efficient search of similar functions in binary code. We apply this approach to identify already known bugs in binaries as follows: starting with a known vulnerable binary function, we are searching for binaries containing the same vulnerable function, which are possibly compiled for different architectures and compiler optimizations. To this end, we identify a set of code features that do not vary a lot across different compilers, optimization options, operating systems, and CPU architectures. These features can be grouped in two categories: *structural features* and *numeric features*. The structural features denote the structure of control-flow inside a binary represented by its CFG. Our experiments show that structural information is a very robust feature for function similarity. However, it is computationally expensive and impractical for comparing a large number of functions. To remedy this situation, we use the second type of features. Numeric features represent meta information about a binary function, e.g., the number of instructions and number of basic blocks. We embed these numeric features in a vector space and leverage techniques from machine learning to quickly identify a set of candidates to be checked for structural similarity. We implemented a prototype of our approach called discovRE. Based on the implementation, we evaluate the correctness of our method and show that it is four order of magnitude faster than the state-of-the-art academic approach for cross-architecture bug search in binary code. Also, based on a vulnerable function extracted from x86 OpenSSL code, we show that we can find the *Heartbleed* vulnerability in an Android system image that contains over 130,000 native ARM functions in less than 80 milliseconds.

In summary, we make the following contributions:

- We present a novel, multistaged approach for *efficient* and *robust* search of binary functions in large code bases. The key property of our approach is that it works across different CPU architectures, operating systems, and compiler optimization levels.
- We propose a set of robust numeric features that allow for a fast similarity comparison between binary functions.
- We introduce a new metric to measure structural similarity between two binary functions based on the maximum common subgraph isomorphism (MCS). To reach efficiency, we use a very good approximate and efficient solution to the MCS problem.
- We implement a prototype of our approach called discovRE, and evaluate its performance against state-of-the-art approaches.
- We demonstrate the application of discovRE for cross-architecture vulnerability search in real-world applications.

II. Approach Overview

We focus on a similar use case to the one presented by Pewny et al. [53]. That is, we start from a known bug (vulnerable function) and then search for the same vulnerability in other binaries by identifying similar functions in those binaries. A high-level overview of discovRE is presented in Figure 1. It takes as input a code base of known functions and a binary function to be searched in the code base. We use IDA Pro [6] to disassemble the binary code and then extract numeric and structural features that we use to compute similarity. Examples for numeric features are the number of instructions, size of local variables, and number of basic blocks (§III-B). Structural features include the function's CFG and other features of its basic blocks. Should the binary be obfuscated several methods can be used to extract the binary code (e.g., [39], [60], or [65]). For each function the code base stores the set of extracted features.

The similarity computation comprises two filters: a *numeric filter*, and a *structural filter*. These filters have increasing precision, but also increasing computational complexity. For this reason we have combined them in a pipeline so that we can filter out dissimilar functions at an early stage. Thus, we use the complex filter only on a small set of candidate functions.

Numeric filter. The numeric features of the searched function serve as search pattern in the code base. We use the k-Nearest Neighbors algorithm (kNN) to find similar functions based on these features. Here, we use the observation that numeric features can be compared efficiently. To this end, we identified a set of features that enables a reliable comparison of functions across different architectures.

Structural similarity. This is the most precise yet most complex filter. This filter checks the similarity of the CFG of the target function against the set of candidates that passed the previous two filters (§III-C). To this end, we present a similarity metric based on the maximum common subgraph isomorphism (MCS). Although the MCS problem is NP-hard, there exist efficient solutions that achieve a very good approximation.

III. discovRE

In this section we describe discovRE in details, focusing on the techniques that it uses to find similarities between binary functions. In order to identify a robust set of features, i.e., features that do not change or change only slightly over different compilers, compilation options, operating systems and CPU architectures, we need a code base that allows for a substantial analysis. Hence, we start by describing the data set that was used to derive robust features and several key parameters.

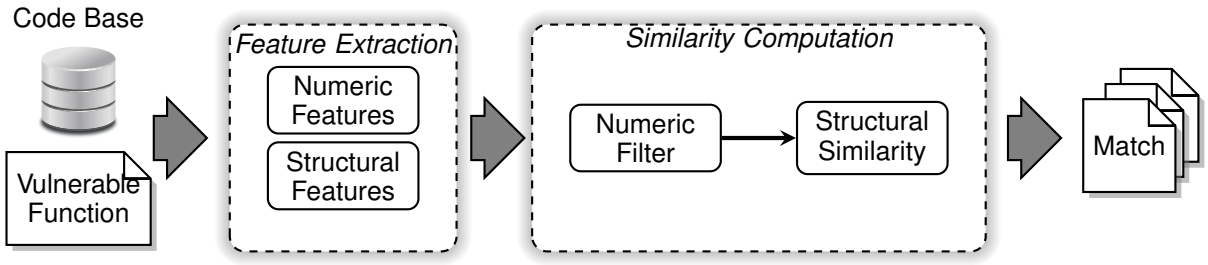


Fig. 1: discovRE architecture

A. Data Set

To derive robust features, we needed a large collection of binaries that are compiled with a variety of compilers, compilation options and for different CPU architectures and operating systems. Further, in order to match functions from different binaries, a ground truth is required at binary function level. This is possible by keeping the symbolic information in the binary. To the authors' best knowledge, no such collection exists, and we therefore created it ourselves. To this end, we selected four widely used compilers, three CPU architectures, and two operating systems. Seven well-known open source projects were compiled with above combinations where feasible with a total of over 1,700 unique compiler options. In the remainder of this section, we present the details of our data base.

For personal computers and server systems, the most prevalent CPU architectures today are Intel-compatible x86 and x64 CPUs. In the world of mobile computing, the ARM architecture is the most common one. The design philosophy behind both CPUs is different. While Intel CPUs offer a rich instruction set (CISC architecture), ARM CPUs are designed to support a limited set of CPU instructions (RISC architecture). A more detailed discussion about the differences is out of the scope of this work. From various revisions, the ARM v7 CPU was selected for its wide distribution in consumer hardware. For x86 and x64, we allowed the compiler to utilize extended instruction sets, such as MMX and SSE.

To cover a vast spectrum of compilers, we used four of the most widely used compilers. Namely, the compilers GNU GCC version 4.8.2 (GCC)[25], LLVM Clang version 3.3 (CL)[41], Intel ICC version 14.0.1 (ICC)[31] and Microsoft Visual C++ version 16.00 (VC)[48] were chosen. All selected compilers support the Intel x86 and x64 architectures. GCC and CL additionally support the ARM architecture. We compiled Windows binaries with GCC (MinGW), ICC, and VC. Linux binaries were compiled with GCC, ICC, and CL.

Regarding compiler optimization options, we only set one explicit limit. We excluded the optimization *function inlining*. Inlining replaces the function call to small functions by the actual code of that function. Thus, the resulting binary lacks a function call at this location for the benefit of improved runtime, however, at the cost of slightly larger binary size. Function inlining would have interfered with establishing a ground truth, as there would emerge a code overlap in the corresponding function labels.

	Windows				Linux		
	GCC	CL	ICC	VC	GCC	CL	ICC
Intel x86	166	-	98	120	166	83	98
Intel x64	166	-	98	120	166	83	98
ARM v7	-	-	-	-	166	83	-

TABLE I: Number of compilation options per compiler and CPU architecture

From the remaining compiler optimization options, we selected a wide spectrum. Alongside standard compiler options that have no special optimizations or optimize for speed or size, options that aim at specific optimizations, such as loop unrolling, were taken into account. Table I shows the number of different compilation options per compiler and CPU architecture. There are altogether 1,711 different compiler options.

1) Open source projects: The evaluation requires the collection of source code to meet several conditions: firstly, the source code needs to be compiled on different operating systems and CPU architectures, secondly, the source code has to be compatible with the selected compilers.

We selected BitDHT [1], GnuPG [2], ImageMagick [30], LAME [57], OpenCV [32], SQlite [3], and stunnel [4] for our experiments. They contain a total number of over 31,000 functions that implement a wealth of different algorithms. All libraries are widely used and hence serve well as testing body.

When compiling for different operating systems or CPU architectures, it has to be made sure that the underlying source code remains the same. During the compilation process, the preprocessor checks for a set of definitions that aim at distinguishing the corresponding operating system or CPU architecture, e.g., `#ifdef WIN32`. The scrutinized open source projects contain several code portions that are specific to a certain operating system. These functions were removed from the data set, as they implied changes of the source code with inevitable changes in the resulting binary. We identified and excluded 1,158 functions that contained such OS-specific code. Additionally, some functions contain highly optimized assembler code. Assembler code is not affected by compiler optimizations and thus the resulting machine code remains the same over all compiliation options. As the outcome of the evaluation over these binary functions would be quite clear, we decided to remove them.

We observed that some compilation options result in the same binary code. To identify binary duplicates, it does not suffice to compare the binaries by their checksums, as some compilers insert the compilation time stamp into the binary. Instead, we used ssdeep with a minimum similarity score of 95. This procedure reduced the number of unique binaries from over 1,700 to about 600 for each corresponding set of binaries. This means that many compilation options seem to have virtually no effect on the resulting binaries.

By leaving the symbolic information intact during compilation we could match functions from different binaries by their name. The resulting machine code is not altered by keeping symbols, as this information resides either in a different part of the executable or is stored in an external file, depending on the compiler and compilation options. Special care needed to be taken with idiosyncrasies in the compiler name-mangling schemes to match the symbolic names.

All binaries were disassembled and the features were extracted for each function. The commercial disassembler IDA Pro [28] was used as framework. It required little implementation effort, as it natively supports the analysis of a vast number of different CPU architectures. Additionally, IDA Pro is able to load symbolic information from different compilers. We only considered functions with at least five basic blocks in our experiments. This is unproblematic as the probability that less complex functions have a vulnerability is significantly lower [46].

With all above considerations, 8,895 individual functions with at least five basic blocks could be matched by their symbol name over all compilation options. This resulted in over 6 million individual binary functions.

2) Removal of duplicate functions: Exact binary duplicates obviously have the same numeric and structural features. As these would lead to a bias during the robustness analysis of the features, we first needed to exclude them. To this end, we calculated the checksum of each function over its bytecode sequence. In the bytecode all references to memory locations were zeroed as they are likely to change over compilations. This includes target addresses of calls, string references and other pointers such as virtual method tables.

Despite careful removal of duplicates at file level we still observed that a large amount of functions had the same checksum. After eliminating the possibility of a hash collision from different inputs, we could think of two possible reasons for this phenomenon: Either the source code had been duplicated, i.e., there exist two different functions with the same source code, or the source code had been compiled to the same machine code, despite different compilation options.

We checked both hypotheses and found both to be true. While the duplication of source code amounted for 1.41 % of the total code base, the lion's share were compilation duplicates with a total amount of 58.06 %. Although very different compilation options had been selected, they seemed to have only limited effect on the resulting binary code. This insight is especially interesting for labeling methods based on the binary code, such as IDA FLIRT [28].

3) Identification of robust numeric features: Each function holds a wealth of numeric information or "meta data", e.g., the number of instructions. For the numeric filter, the key hypothesis is that there exists a set of numeric features that can correctly characterize a binary function, even across above described boundaries. The intuition is that a function has semantics, which is expressed in the source code by a certain syntax. The compiler translates the function's source code to a target platform, where it can use several optimizations. However, the underlying semantics must remain the same. Thus, certain key features should not alter significantly.

As an extensive description of all scrutinized features is outside the scope of this work, we only concentrate on the most interesting ones. Quite obvious is the number of instructions, the size of local variables and the number of parameters. We additionally classified each instruction by its functionality [33]. The classes are arithmetic, logic, data transfer, and redirection instructions. Based on the CFG, we counted the number of basic blocks and edges. Additionally, we counted the number of strongly connected components as a rough estimate for loops. By analyzing the call graph we counted the number of incoming and outgoing edges to/from a function as a measure how "often" a function is called and how many functions it calls.

There are two key aspects to assess the robustness of a numeric feature: Firstly, its value should only change minimally over all compilation options, and secondly, the values should be distributed over a larger domain.

To assess the quality of the former, we employed the Pearson product-moment correlation coefficient. A value of -1 resembles perfect anti-correlation of two numeric vectors, a value of 0 means no correlation, and a value of +1 denotes perfect correlation. Here, we are seeking correlations towards +1. To generate the numeric vectors, we matched the symbolic names of the functions over the according binaries and extracted the scrutinized feature. We then calculated the average over all pair-wise correlation coefficients and additionally the standard deviation. The quality of the value distribution was checked by counting the number of different values and calculating the standard deviation.

Table II shows the robustness of various features. Higher values in the first two columns imply better robustness of a selected feature. Column three displays the average correlation, column four shows the standard deviation of the correlation. We aim for a high average correlation and a low standard deviation. Highlighted in Table II are the features we selected as sufficiently robust.

B. Numeric filter

To minimize the expensive structural comparison of two CFGs, we added an efficient filter by only relying on numeric features. We chose the kNN algorithm, as it fits best for a candidate search. The filter uses the robust numeric features described above to efficiently identify a set of candidates. In this section, we will present the details of the filter and discuss potential alternatives.

There exists a vast amount of machine learning algorithms that are apt to search a function in a code base by their numeric features. However, the search for similar functions has some special requirements that allow us to sort out some machine

Feature	sd(values)	values	avg.cor	sd(cor)
Arithmetic Instr.	39.483	623	0.907	0.109
Function Calls	22.980	273	0.983	0.073
Logic Instr.	49.607	625	0.953	0.067
Redirections	40.104	556	0.978	0.066
Transfer Instr.	163.443	1,635	0.961	0.075
Local Vars.	2.78E6	890	0.983	0.099
Basic Blocks	48.194	619	0.978	0.067
scc	25.078	389	0.942	0.128
Edges	76.932	835	0.979	0.066
Incoming Calls	46.608	261	0.975	0.086
Instr.	295.408	2,447	0.970	0.069
Parameters	2.157	38	0.720	0.228

TABLE II: Robustness of numeric features. The selected features are highlighted.

learning algorithms. One key requirement is performance. Hence, to make the system feasible for real-world applications, the search algorithm needs to find similar elements in a large amount of elements in a few milliseconds. Memory consumption should be moderate to cope with large code bases. The classification algorithm should be able to return multiple elements in one search. Further, the algorithm needs the ability to cope with many different labels, as the number of functions in the database is roughly the same size as the number of labels.

The machine learning algorithm being suited best for all aforementioned criteria is the kNN algorithm and was hence chosen as numeric filter. a potential alternative are Support Vector Machines, however, they require relatively much memory, additionally, the setup times are much higher. The main benefit of kNN is that the list of the k most similar functions can be retrieved. Next, the control flow graphs (CFGs) of the resulting candidate functions are compared to that of the queried function.

1) Data normalization: The numeric features have different value ranges. For example, the size of local variables commonly ranges from zero to a few thousand bytes, while the number of arguments is in the range of zero to about a dozen. This would lead to problems in the distance calculation of the kNN algorithm. Hence, each feature is normalized, i.e., its value domain is adjusted to an average value of 0.0 and a standard deviation of 1.0. Both statistic values were derived from the data set.

It is well possible that two functions collide wrt. their numeric features, i.e., the features have the same values. During the code base creation phase, duplicate entries are mapped to the same point in the vector space. If that representative point is returned by the kNN algorithm, all functions that are mapped to that point are returned. Additionally, a distance threshold is introduced to remove functions that are too dissimilar.

2) Evaluation of the numeric filter: There exists a plethora of different implementations of the kNN algorithm. We selected OpenCV's Fast Library for Approximate Nearest Neighbors (FLANN) [50]. It brings a variety of different nearest neighbor implementations, of which we selected three different that each highlight special aspects:

- linear index,
- k-d trees, and
- hierarchical k-means.

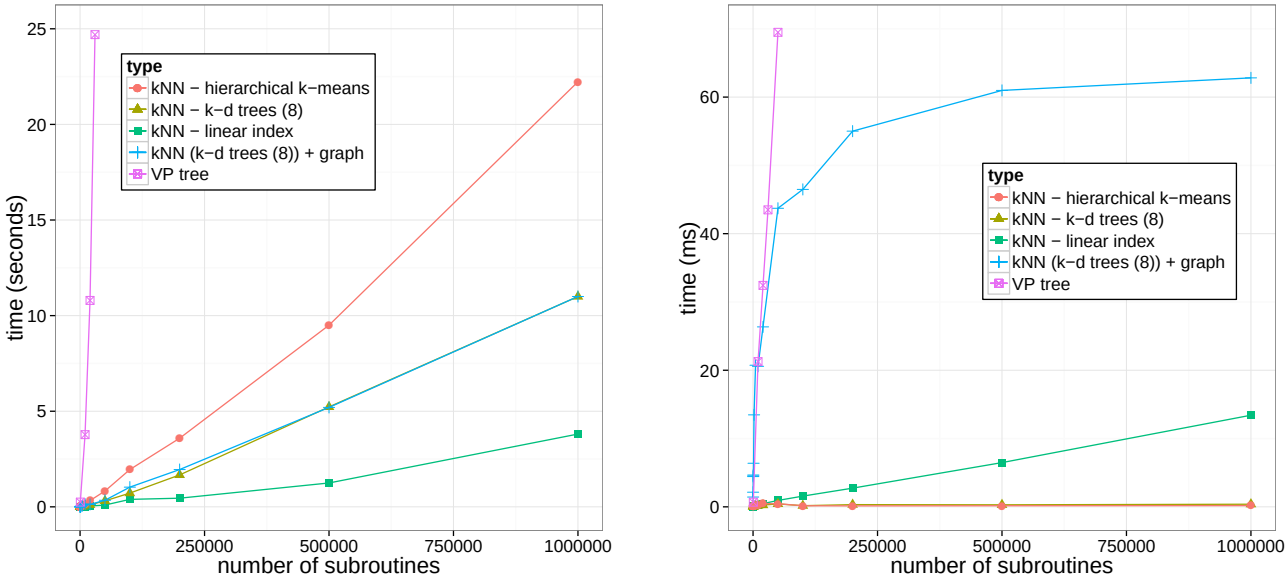
The linear index iterates over all elements of the search space and stores the current nearest neighbors in a priority queue of size k. Once all elements have been visited, the queue contains the k nearest elements. Hence, the query times are in $\mathcal{O}(n)$. As the algorithm does not need any special data structures for storage, the memory consumption is negligible. We chose this algorithm to serve as lower/upper bound to give an estimate about the expected times. The k-d trees algorithm is a set of multidimensional binary search trees. During the creation of one tree, one dimension is randomly chosen and the data set is subdivided by the hyperplane perpendicular to the corresponding axis. In the query phase, all trees are searched in parallel [50]. For low dimensions, the average running time of a query is $\mathcal{O}(\log n)$. Clearly, this algorithm requires more memory, as it needs to store several k-d trees, which is typically in $\mathcal{O}(kn)$, with k being the number of trees. We set the number of trees in the k-d trees algorithm to 8 as a compromise between quality and memory consumption. Hierarchical k-means recursively splits the data set into smaller clusters, aiming at maximizing the distance between the clusters. Also here, the expected running time of a query is in $\mathcal{O}(\log n)$. The space complexity is typically in $\mathcal{O}(k+n)$

One key observation from the data set was that some optimization options generally tend to generate more machine code instructions per line of source code. This means that we can expect larger absolute differences for larger functions. We responded to this phenomenon by applying the decadic logarithm to all numeric values. After normalizing each feature, we populated the vector space of each kNN algorithm and measured the times to create the search structure. After that we let the kNN implementation search for a randomly chosen point. We repeated the search 10,000 times and calculated the average values where feasible.

Note that the classical application of the kNN algorithm performs a majority vote on the labels of the k nearest neighbors. Contrary to that, we actually submit the k nearest points to the detailed filter stage. Figure 2 shows the creation and query times for various numbers of points. Most notably, k-d trees take significantly longer to set up. Interestingly, the query times do not significantly increase over the number of functions in the search index, apart from the linear index.

The evaluation shows that even for larger code bases (e.g., a full Android image has over 130,000 functions) both, the times to create the search structure of the code base and the times to search similar functions in it are feasible for real-world applications. However, there still remains space for further optimizations, as we show in the next section.

3) Dimensionality reduction: As higher dimensions of a data set might lead to significantly longer runtime of the kNN algorithm (see the curse of dimensionality [45]), we analyzed the data set for linear dependencies using principal component analysis. With five components the cumulative proportion of the variance was already at 96.82 %. We repeated above measurements on the data set with reduced dimensionality only for k-d trees as selected pre-filter. By reducing the



(a) (b)

Fig. 2: (a) setup and (b) query times for the presented algorithms.

dimensionality of the search space to 5, the average setup time was reduced by 3.1 %, and memory consumption was lowered by 4.6 %. Most interestingly, query times dropped by 31.1 %, which gives a significant advantage to repeated queries on the same data structure.

4) Conclusion: Based on the above results, we decided to use the kNN algorithm based on k-d trees. We have found that the value $k = 128$ is sufficient for our purposes. Note that this value is configurable and can be adapted.

C. Structural Similarity

In this search stage, the CFG of the queried function and the CFGs of the candidates from the kNN search are compared. The CFG is an easily approachable structure and has been shown to be a good predictor for the labeling of binaries [29] and was hence selected as structural feature. There exist many other potential syntactic and semantic features, e.g., the abstract syntax tree, the decompiled source code, or semantic approaches. However, we deem them too complex, as they require a large amount of preparation time.

Input to the complex filtering stage are the nearest neighbors of the queried function. During this stage, a linear search over all elements is conducted. We define the (intra-)procedural control flow graph as a structural representation of a function f given by basic blocks and their connections [13]:

$$G_f := (\{v | v \text{ is basic block in } f\}, \{(u, v) | u \text{ redirects to } v\}).$$

A basic block is a sequence of machine code instructions that has exactly one entry and one exit point. Commonly, a redirection is a conditional jump, a function call or a return from a function. The CFG represents the program logic of a function and consequently should not alter substantially over compilers and optimization options. Prior work [24, 26] showed that features based on the CFG perform very well in labeling functions.

1) Basic block distance: We not only rely on the structural properties of the function. Additionally, we enrich all nodes with several features from their respective basic blocks. First, each node receives a label according to its topological order in the function. Second, string references and constants are stored. Third, each node contains its own set of robust features. We assumed that robust features of functions closely resemble to those of basic blocks, and hence used a similar feature set, sans features that are not applicable at basic-block level, such as the number of basic blocks.

For each numeric feature, we calculate the absolute difference and multiplied a weight α_i to it. The resulting distance function is:

$$d_{BB} = \frac{\sum \alpha_i |c_{if} - c_{ig}|}{\sum \alpha_i \max(c_{if}, c_{ig})}$$

with c_{if} being numeric feature i of function f. For lists of strings, the Jaccard distance is used as distance measure. The term in the denominator ensures that the distance function is in the range $[0, 1]$. In Section III-C4 we will discuss good choices for α_i.

2) Graph matching algorithm: Several approaches have been described to calculate the similarity of two CFGs. A prominent approach uses bipartite matching on the list of basic blocks [29]. As the bipartite matching algorithm is agnostic to the original graph structure, the algorithm was adapted to respect neighborhood relations. One drawback of the described method is that an erroneous matching might propagate and thus lead to undesirable results. The distance function between two basic blocks is the weighted sum over their respective features.

A different approach is a graph similarity algorithm based on the maximum common subgraph isomorphism problem (MCS) [47]. It is a well-known problem to finding the largest subgraph isomorphic to two graphs.

The traditional MCS distance function is defined as

$$d_{mcs.orig}(G_1, G_2) := 1 - \frac{|mcs\,(G_1, G_2)|}{\max\left(|G_1|, |G_2|\right)}$$

with graphs G_1, G_2. To account for the similarity between two basic blocks, we extended the distance function to the following form:

$$d_{mcs}(G_1, G_2) := 1 - \frac{|mcs\,(G_1, G_2)| - \sum d_{BB}\,(b_i, b_j)}{\max\left(|G_1|, |G_2|\right)}$$

As the problem is NP-complete for arbitrary graphs, special care has to be taken to keep the complexity as low as possible. Our implementation uses the McGregor algorithm [47]. In the expansion phase candidate nodes of both graphs are checked for potential equivalence and selected as match if they are considered equivalent. At this stage we use the basic block distance d_{BB} for two purposes: early pruning and deciding which candidate pair to expand next. A pair of basic blocks is removed from the set of potential matches if the distance exceeds a certain threshold. Our experiments showed that 0.5 is a good threshold. Next, the candidate pairs are sorted by their distance. The pair with the lowest distance is expanded first, as it is highly likely that nodes with the lowest distance are indeed a correct match.

To cope with the potentially exponential running time, we terminate the MCS algorithm after a predefined number of iterations and return the currently calculated minimal distance. We call the relaxed form the maximal common subgraph (mCS).

3) Evaluation of the structural similarity: To evaluate the function distances, we first needed to derive a parameter set α_i for the basic block distance d_{BB}. On that basis, we could scrutinize the convergence speed of the graph-based approaches. After that, we could measure the predictive quality of all presented approaches.

4) Evaluation of the basic block distance: Our goal was to find a parameter set for the basic block distance function d_{BB} (cf. Section III-C1). However, even with debug information there is no straightforward way of matching a piece of source code to its corresponding basic block. Especially with more aggressive optimization options whole portions of source code might be removed (dead code elimination), or even duplicated (loop unrolling). Hence, we cannot establish a ground truth at the basic-block level to match two basic blocks from different binaries.

The data set, as described in Section III-A, already contains a ground truth about the labels of each function. Assuming different functions imply (a large amount of) different basic blocks, and simultaneously same functions imply same or similar basic blocks, we can rephrase above problem. We seek a parameter set that maximizes the distance of different functions while simultaneously minimizing the distance of equal functions. Put another way, we want to maximize the difference between equal and different functions:

Feature	α_i
No. of Arithmetic Instructions	56.658
No. of Calls	87.423
No. of Instructions	40.423
No. of Logic Instructions	76.694
No. of Transfer Instructions	6.841
String Constants	11.998
Numeric Constants	15.382

TABLE III: Best parameter set for distance function d_{BB}.

$$\max\left(d_{BB}\left(f_i, g_j\right) - d_{BB}\left(f_i, f_j\right)\right),$$

with $f \neq g$ being functions with different labels.

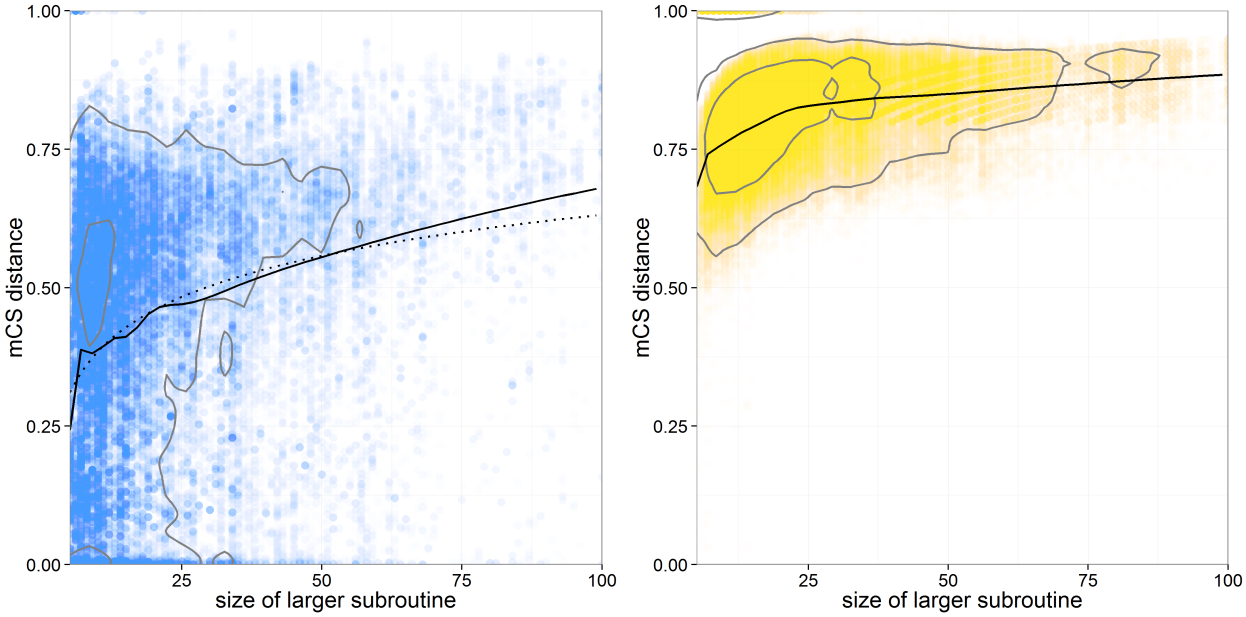
To approach this optimization problem we used a genetic algorithm. Our implementation uses GALib [58], a widely used library for genetic algorithms. We executed an arithmetic crossover using a Gaussian mutator for 100 times. The population size was 500 and we let it run for 1,000 generations. The value range for α_i was set to [0..100]. For the calculation of equally-labeled functions, two random compilation options and two functions with the same label were drawn. Note that we performed this experiment on all binaries, i.e., over all compilers, optimization options and CPU architectures. The approach was similar for differently-labeled functions, but additionally we allowed to draw a function from the same binary. We only made sure that the function label was different.

We scrutinized the ten highest scoring parameter sets. It showed that *strings* and *numeric constants* were assigned a relatively low weight by the genetic algorithm. One possible explanation is that by far not every basic block contains constants and thus the data is deemed less important. Over all runs, the *number of data transfer instructions* received a relatively low weight. Most probably the reason is that transfer instructions are very common and thus do not add much information. The highest values were given to the *number of logic instructions* and the *number of function calls*. The *number of instructions* received values in the middle of the value range. The overall best parameter set is depicted in Table III. The calculated average distance between different and equal functions for this parameter set is 0.378.

5) Robustness of the Graph Distance Function: In this section we evaluate the predictive quality of the mCS algorithm. For that, we calculate the distance between randomly selected equal and different functions. To keep the runtime manageable, we abort the mCS algorithm after 10,000 iterations and return the current maximal value.

In Figures 3 a) and b) the size of the larger function is depicted on the x-axis, the distance is shown at the y-axis. The average distance of functions with the same label, Figure 3a a), is 0.436 (standard deviation 0.242). The average distance of different functions, Figure 3b b), is 0.814 (standard deviation 0.081).

Each point is plotted with transparency, hence the opacity of a coordinate gives an indicator about the respective density. Additionally, topological lines depict the density at a given



(a) Equal Functions

(b) Different Functions

Fig. 3: Scatter plot showing distance function d_{mcs} for the set of functions.

area. In average, equal functions show substantially lower distances than different functions.

Comparing both figures it becomes evident that the graph distance function $d_{mcs.orig}$ is robust.

6) Uncoupling the distance function from function size: As seen in Figure 3, the mCS distance measure has a tendency towards higher values for larger graphs. This tendency is not surprising, as larger graphs have a higher potential of adding more constraints. E.g., some compilers optimize towards a single returning basic block whereas other compilers create code that allows returning from several different locations.

An approximation of the distance function to a constant has the advantage that a constant threshold value can be used to distinguish equal from different functions. This in turn allows us to utilize the distance function in efficient search algorithms.

Figure 3a shows two regression lines in each graph. The solid line is a B-spline crossing central midpoints. It is computationally more expensive than other methods, but gives a good estimate about the ideal regression line. It has a relative standard error (RSE) of 0.164. The dotted line depicts a logarithmic regression with an RSE of 0.168.

To uncouple the distance function from the number of basic blocks, the distance function d_{mcs} is altered in the following way to compensate for the function size:

$$d_{mcs\_comp}(G_1, G_2) := \frac{d_{mcs}(G_1, G_2)}{comp\,(G_1, G_2)}$$

with $comp$ being the logarithmic compensation term:

$$comp\,(G_1, G_2) := i + k \log\left(\max\left(|G_1|, |G_2|\right)\right).$$

7) Convergence Speed of the MCS: The potentially exponential runtime of an exact solution for the maximum common subgraph problem deems the problem unfeasible for real-world application. The computational complexity of a good approximation, however, may be significantly lower. Hence, an important research question is when to terminate the mCS algorithm, or how fast the mCS algorithm converges to its maximum value. We conducted the following experiment only on the basic-block sensitive approach d_{mcs}.

Our mCS implementation provides a callback mechanism that is invoked when a new solution is found, allowing to collect information about the currently found subgraph. As discussed in Section III-C2, the basic block distance function d_{BB} serves as good heuristic for the mCS algorithm in order to early prune the search space. At this place, we can discard a potential match if d_{BB} is too high.

Figure 4 depicts the advance of the currently found maximal common subgraph size of the first 1,000 iterations for an example function. After a steep, quasilinear ascend, the mCS is reached after only 56 iterations. We conducted an analysis of the convergence speed over our data set. In average 64.68 iterations are needed for the algorithm to generate a common subgraph of the same size as the maximal size after 10,000 iterations. When aborting the mCS algorithm after $16 \max\left(|G_1|, |G_2|\right)$ iterations, the size of the largest found subgraph reaches on average 99.11% of the subgraph size after 10,000 iterations.

As the exact value of the MCS is no precondition to correctly label functions. The potentially exponential complexity can instead be reduced to a linear runtime with a reasonable approximation to the MCS.

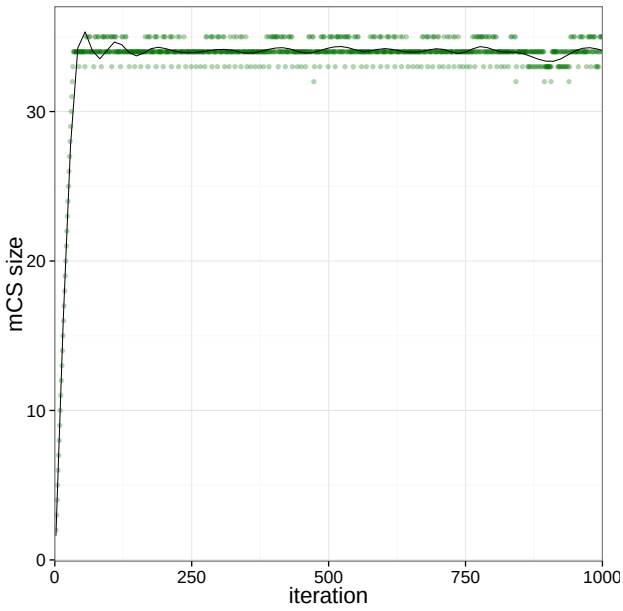


Fig. 4: Advance of the current maximal subgraph of function `pkey_rsa_ctrl` compiled with options CLANG_01 and CLANG_02.

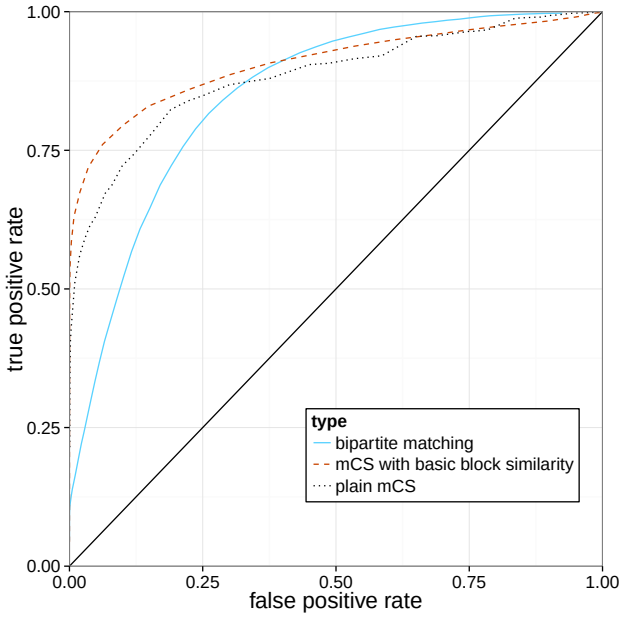


Fig. 5: ROC curves of different distance measures based on the CFG.

8) Comparison of d_{mcs} with Other Distance Functions: In Section III-C2 we described three different algorithms to match two CFGs. The first one is based on bipartite matching, as described in [29], the second and third algorithms are both based on the MCS: The plain mCS algorithm $d_{mcs.orig}$ and the extended algorithm d_{mcs}, using the basic block distance d_{BB} as additional measure.

Figure 5 shows the receiver operating characteristics (ROC) of the three distance functions with logarithmic compensation. To calculate the characteristics, we computed the distances of 1 million randomly chosen pairs of equal and different functions. We compared the classic mCS distance function $d_{mcs.orig}$ with the basic-block sensitive approach d_{mcs} and with the bipartite matching distance [29] The figure shows that the d_{mcs} performs better than the plain mCS algorithm. The ROC curve of the bipartite matching approach is significantly lower than that of both structure-aware approaches for low false positive rates.

During the evaluation we also measured the respective timings. The mCS algorithm based on $d_{mcs.orig}$ needed in average 6.3 ms per function, the improved mCS algorithm based on d_{mcs} needed only 4.4 ms, while the bipartite matching approach needed 4.2 ms.

Thus, the running times of both approaches are in the same order of magnitude, however, the quality of our approach is significantly higher.

9) Comparison to VP Tree: Bunke showed that the maximum common subgraph distance measure d_{mcs} on the basis of the MCS is a metric [18]. Hence, it is possible to use efficient search structures, such as vantage point (VP) trees [64] to efficiently search the most similar binary function. The usage of VP trees would make the numeric filter obsolete, as the function CFG can be directly and efficiently searched.

A VP tree is created by selecting one element as pivot element. All distances between the remaining elements and the pivot element are computed. The set of remaining elements is split up into two sets of roughly the same size based on their distance to the pivot element. The elements closer than the selected threshold are added to the left branch of the pivot element, the other elements are added to the right branch. This procedure is recursively repeated until the set size becomes sufficiently small.

To find the nearest neighbor of a given element in the VP tree, the distance between the pivot element (the respective root node) and the search element is computed. The tree is appropriately traversed by making use of the triangular inequality of the metric until the nearest neighbor or set of neighbors is found.

In the average case, VP trees have a logarithmic search complexity. This comes, however, at the cost of building the VP tree and therefore is only feasible when querying a larger number of elements.

Also depicted in Figure 2 are the creation and search times for a VP tree. It clearly shows that both, the creation and search times, are significantly higher than the timings of the presented approach. One reason for the long creation time of might be a high number of rebalancing steps. These pose a critical point during the creation phase. Both timings make a solution based on VP trees infeasible for real-world applications.

IV. Evaluation

In this section, we describe the results of the experiments we have performed to evaluate discovRE. We first evaluate

the correctness of our similarity comparison approach. Then we show the efficiency achieved by combining the numeric and structural filters. Finally, we apply discovRE to find bugs in binaries across several architectures.

A. Similarity Metric

To evaluate our similarity metric, we performed an extensive evaluation using OpenSSL. Note that OpenSSL was not included in the set of binaries that we used to derive code features. For our evaluation, we compiled OpenSSL (version 1.0.1.e) for x86, MIPS, and ARM with different compilers, compiler options, and operating systems as explained in Section III-A. This resulted in 593 binaries with 1,293 sufficiently large functions. We kept the debug symbols as they provide a ground truth and enable us to verify the correctness of matching using the functions symbolic names. For clarity, we denote by $f_n^{c,a}$ the nth function in OpenSSL that was compiled for architecture a and using compiler setting c. c comprises type and version of compiler, compiler optimizations, and the target operating system. To test discovRE for matching a given function n, we:

1) randomly select a target function $f_n^{c1,a1}$ to be searched for,
2) randomly select a matching function $f_n^{c2,a2}$,
3) randomly select a a set F of 99,999 functions, and
4) construct the code base to be queried as $C = F \cup \{f_n^{c2,a2}\}$ ($|C| = 100,000$).

We then used discovRE to query for $f_n^{c1,a1}$ in C. The ideal result would be $f_n^{c2,a2}$. To include more variety in the code base, we included functions from the data set described in Section III-A. We repeated the previously described process 1 million times. selecting n at random with each iteration. Table IV shows the results of this evaluation. At a high level, in 93,93 % of the cases the matching function was in the set of 128 functions returned by the numeric filter (kNN). When using the numeric filter alone, the matching function was on average at rank 5.5. Interestingly, when combining that with the structural filter, the matching function was *always* at the first rank. The average query time was 56.48 ms, illustrating the efficiency of discovRE. A closer analysis of the false predictions revealed that in most cases incorrect matching was caused by the loop unrolling compilation option. The unrolling of loops duplicates loop bodies several times. Hence, several numeric features are significantly altered, which in turn increases the distance.

B. Cross-Architecture Bug Search

In this section, we apply discovRE to real-world case studies of recent and prominent vulnerabilities. Here we base our experiment on the techniques used by Pewny et al. [53]. This evaluation searched for the *Heartbleed* vulnerability in OpenSSL binaries from three sources:

1) self-compiled binaries for x86, MIPS, and ARM
2) the Linux-based router firmware DD-WRT (r21676) compiled for MIPS [10]
3) a NAS device (Netgear ReadyNAS v6.1.6) with an ARM processor [11]

We contacted the authors of [53] and they kindly agreed to share the binaries used in their evaluation. We very much appreciate this good scientific practice. This way, we could ensure that both approaches are tested on the same binary code base. All evaluations have been executed on a commodity PC with an Intel Core i7-2720QM CPU with 2.20 GHz and 8 GB DDR3 RAM on a single CPU core. We choose this system because it is very similar to the setup used in their paper (even a bit slower).

We also extend the evaluation performed by Pewny et al. by performing bug search over all binaries of the firmware images. In their original evaluation, the authors first extracted the OpenSSL libraries from the firmware images and limit the bug search to these binaries. We conduct a more realistic experiment where we do not assume that we know the vulnerability exists in a given binary from the firmware image. This is a strong assumption given that firmware images may contain hundreds of binaries, making the problem of identifying the vulnerable binary very challenging. For example, the ReadyNAS image contains 1,510 binaries. Moreover, this is especially true if static linking is used, where library code is copied into the target executable. For better comparability, we report the results on both the OpenSSL binaries and the whole binaries contained in the firmware image. Also, the authors reported the normalized average time needed for the preparation and query steps. This is defined as the average amount needed to process 10,000 basic blocks. This allowed us to extrapolate the absolute times needed by their approach to perform bug search on the same firmware images.

We identified a buggy function in the respective binaries, namely we chose CVE-2014-0160 (Heartbleed). The Heartbleed bug manifests itself in the two functions `dtls1_process_heartbeat` (DTLS) and `tls1_process_heartbeat` (TLS) that are virtually identical. Thus, the search for either of the functions should also return the other function in the list of similar functions.

Table V shows the results of the comparison of the approaches by Pewny et al. and discovRE. While the Multi-MH method has some problems in correctly identifying TLS from x86 in DD-WRT or DTLS from x86 in MIPS, Multi-k-MH scores a lot better. In only one case, when trying to match the x86 TLS function in DD-WRT it ranks the correct function only at place 5. In contrast, discovRE always correctly identifies the searched function(s). Summarizing, the quality of discovRE slightly supersedes that of of Multi-k-MH.

One crucial unit of measure is the time needed to execute a bug search, especially for real-world problems. Pewny et al. state that the normalized running time to prepare the data was either 18.7 minutes (Multi-MH) or 286.1 minutes (Multi-k-MH) per 10,000 basic blocks. We measured the overall time of our approach, i.e., including the time IDA Pro needed for the disassembly and its analysis in order to be more realistic. Our approach needs only 8.7 seconds for 10,000 basic blocks, which is a speedup by two or three orders of magnitude, depending on the approach.

In a typical scenario a binary is only prepared once and then stored in a code base. Typically, there would several hundreds or even thousands of binaries reside in the code base. Once a new bug is published, the vulnerable function is identified and queried in the code base. Thus, the far more interesting time is the actual bug search. Here, the approach

Method	Percent Correct	Rank	Query Time
kNN – k-d Trees (8)	93.93 %	5.50	0.32 ms
discovRE (kNN (k-d Trees (8)) + Graph)	93.93 %	1.00	56.48 ms

TABLE IV: discovRE evaluation with OpenSSL. The table shows the percentage of cases with correct matching, the average rank for correctly matched functions, and the average query times.

	Multi-MH		Multi-k-MH		discovRE	
From → To	TLS	DTLS	TLS	DTLS	TLS	DTLS
ARM → MIPS	1;2	1;2	1;2	1;2	1;2	1;2
ARM → x86	1;2	1;2	1;2	1;2	1;2	1;2
ARM → `DD-WRT`	1;2	1;2	1;2	1;2	1;2	1;2
ARM → `ReadyNAS`	1;2	1;2	1;2	1;2	1;2	1;2
MIPS → ARM	2;3	3;4	1;2	1;2	1;2	1;2
MIPS → x86	1;4	1;3	1;2	1;3	1;2	1;2
MIPS → `DD-WRT`	1;2	1;2	1;2	1;2	1;2	1;2
MIPS → `ReadyNAS`	2;4	6;16	1;2	1;4	1;2	1;2
x86 → ARM	1;2	1;2	1;2	1;2	1;2	1;2
x86 → MIPS	1;7	11;21	1;2	1;6	1;4	1;3
x86 → `DD-WRT`	70;78	1;2	5;33	1;2	1;2	1;2
x86 → `ReadyNAS`	1;2	1;2	1;2	1;2	1;2	1;2
Preparation Normalized Avg. Time	18.7 min		286.1 min		0.14 min	
Query Normalized Avg. Time	0.3 s		1 s		$4.1 \cdot 10^{-4}$ s	

TABLE V: Results for Pewny et al.'s approaches Multi-MH and Multi-k-MH and discovRE for OpenSSL.

by Pewny et al. takes about 0.3 s (Multi-MH) or 1.0 s (Multi-k-MH) to search one basic block per 10,000 basic blocks. In comparison, our approach has a normalized average of 0.41 ms, which is three orders of magnitude faster. This value was calculated with just the OpenSSL library in the code base. Note that due to the nature of our approach, this value can significantly change. E.g., the normalized average query time for the complete ReadyNAS image with roughly 3 million basic blocks is 0.015 ms, which would change the factor by one order of magnitude towards our approach.

1) Cross-Architecture Bug Search in Complete Firmware Images: As noted above, the scenario of manually identifying and extracting potentially vulnerable binaries from a firmware image is a tedious task. A better line of action would be to extract all binaries from a firmware image and automatically store them in the code base. To put above evaluation into perspective, we show the total number of basic blocks of aforementioned firmware images in Table VI. While for example the OpenSSL binary of DD-WRT contains already about 11,000 basic blocks, the accumulated number over all binaries is several orders of magnitude larger.

In addition to DD-WRT and ReadyNAS, we obtained a vulnerable Android ROM image (version 4.1.1) compiled for HTC Desire [56]. The HTC Desire has a Qualcomm Snapdragon CPU with an ARMv7 instruction set. We extracted the file system and found 527 native ELF binaries. A total of 133,205 functions could be identified as sufficiently large and were thus stored in the code base. For DD-WRT, 143 binaries could be identified with a total of 34,884 functions. ReadyNAS contained 1,510 binaries with a total of 295,792 functions. The respective preparation times can be found in Table VI.

Even in a realistic scenario the overall time needed to disassemble all binaries and extract the function features is manageable, e.g., about 30 minutes for a complete Android image.

For each combination of CPU architectures and firmware image we searched the Heartbleed TLS bug and additionally CVE-2014-3566 (POODLE). The POODLE vulnerability is found in the function `ssl_cipher_list_to_bytes`. We searched each vulnerable function 20 times in the database and calculated average values where it was useful. Table VII gives some insights about the search times and placements of the queried function, split up in the two search stages of discovRE. While the kNN algorithm is not always able to assign the queried function a high rank, it is still always among the first k results. In the next stage, the mCS algorithm correctly identifies all buggy functions across CPU architectures and ranks them first.

The query times of the kNN algorithm are under 2 ms and thus very low, compared to the mCS algorithm. For larger firmware images, the number of binary functions that share similar numeric features inevitably becomes larger, as the density increases. Hence, the number of candidate functions for the mCS algorithm increases and so does the overall running

time. With well under 90 ms the time is acceptable in real-world scenarios, even for large code bases.

Our results clearly show that the presented syntactic approach outperforms state-of-the-art semantic approaches wrt. speed while still maintaining the same predictive quality.

C. Comparison to BinDiff

We compared our results with BinDiff [22]. BinDiff is a comparison tool for binary files that matches identical and similar functions in two binaries. It is based on IDA Pro and can compare binary files for x86, MIPS, ARM, PowerPC, and other architectures supported by IDA Pro. BinDiff is not designed to handle our use case, i.e., searching for a single function in a code base. It relies among others on calling relationship of functions to identify matches in two binaries. We emphasize that this comparison is informal, as it is designed to help understand the quality of discovRE in comparison to BinDiff when applied for bug search.

For this comparison, we searched for functions corresponding to POODLE and Heartbleed vulnerabilities taken from an x86 OpenSSL binary in an ARM OpenSSL binary. To create a similar settings to our bug search use case, we removed all functions from the IDA Pro database except for the queried function. Then we used BinDiff to check whether it can correctly match this function in the ARM binary. For POODLE, BinDiff erroneously predicted `ssl_callback_ctrl` as match (with a similarity of 0.03 and a confidence of 0.08). BinDiff also fails in the case of Heartbleed; it returned `dtls1_stop_timer` as a match with a similarity of 0.01 and a confidence of 0.03. On the other hand, discovRE correctly identified the vulnerable functions.

V. Limitations

The identification of vulnerabilities in large code bases clearly demonstrates the capabilities of our approach. Nevertheless, there exist limitations that need to be considered: First, our similarity is purely syntactical and thus cannot handle code that may be heavily obfuscated to avoid similarity detection (e.g., malware). To tackle this problem, one can leverage approaches such as [60, 39, 65, 20] to extract and deobfuscate the binary code and then apply discovRE on the deobfuscated code. Moreover, the binary code of many prominent software products is not obfuscated but instead compiled with different options/compilers/architectures and evolves over time. Our approach is very fast and accurate when handling these cases.

Second, our method operates at the function level and thus cannot be used be with sub-function granularity. However, the vast majority of bugs can be pinpointed to one or a list of specific functions. These functions can be efficiently identified by discovRE and presented to the user for further analysis. Also, our approach needs a precise identification of the functions contained in a binary. Our evaluation showed that IDA Pro delivered satisfactory results in our experiments. To further improve these results, one can leverage the recent and advanced academic methods to recognize functions in binary code such as [16, 55]. These methods employ machine learning techniques to accurately recognize functions across multiple operating systems, CPU architectures, compilers, and compiler options.

Third, function inlining may heavily change the CFG of a program. This will impact the quality of structural similarity and may thus become problematic for our approach. We leave the evaluation of discovRE in case of function inlining for future work.

The functions considered by discovRE need to have a certain amount of complexity for the approach to work effectively. Otherwise, the relatively low combinatorial number of CFGs leads to a high probability for collision. Hence, we only considered functions with at least five basic blocks, as noted in Section IV. The potential for bugs in small functions, however, is significantly lower than in large functions, as shown in [46]. Hence, in a real-world scenario this should be no factual limitation.

VI. Related Work

The development of methods for finding vulnerabilities in software has been in the focus of security research for a long time and several techniques have been proposed. For our discussion of related work, we focus on approaches that aim to measure code similarity and search for already known bugs. There also exist orthogonal approaches that aim at finding previously unknown bugs such as AEG [14] or Mayhem [19]. We will not discuss these approaches further since they have a different goal and employ different methods such as fuzzing and symbolic execution.

At the core of searching for known bugs in software is measuring code similarity. Several works focused on finding code clones at the source code level. Token-based approaches such as CCFinder [38] and CP-Miner [43] analyze the token sequence produced by lexer and scan for duplicate token subsequences, which indicate potential code clones. In order to enhance robustness against code modifications, DECKARD [35] characterize abstract syntax trees as numerical vectors and clustered these vectors wrt. the Euclidean distance metric. Yamaguchi et al. [63] extended this idea by determining structural patterns in abstract syntax trees, such that each function in the code could be described as a mixture of these patterns. This representation enabled identifying code similar to a known vulnerability by finding functions with a similar mixture of structural patterns. ReDeBug [34] is a scalable system for quickly finding unpatched code clones in OS-distribution scale code bases. Contrary to discovRE, these systems assume the availability of source code and cannot operate at the binary level.

Due to the significant challenges associated with comparing binary code, many previous works support a single architecture or make simplifying assumptions that do not hold in practice. Flake et al. [24] proposed to match the CFGs of functions, which helps to be robust against some compiler optimizations such as instruction reordering and changes in register allocation. However, the approach could only identify exact CFG matching and is thus not suitable for bug search. Myles et al. [51] proposed to use opcode-level k-grams as a software birthmarking technique. BinHunt [26] and its successor iBinHunt [49] relied on symbolic execution and a theorem prover to check semantic equivalence between basic blocks. Checking for semantic equivalence limits the applicability of this approach for bug search since it is not

Firmware Image	Binaries (unique)	Basic Blocks	Preparation Time in Minutes		
			Multi-MH	Multi-k-MH	discovRE
DD-WRT r21676 (MIPS)	143 (142)	329,220	616	9,419	2.1
Android 4.1.1 (ARM)	527 (318)	1,465,240	2,740	41,921	28.7
ReadyNAS v6.1.6 (ARM)	1,510 (1,463)	2,927,857	5,475	83,766	51.4

TABLE VI: Preparation times and additional data about three real-world firmware images that are used in the evaluation of discovRE. Note that the times shown for Multi-MH and Multi-k-MH are projected times and not actual measurements.

	Heartbleed (TLS)				POODLE			
		Query Time				Query Time		
From → To	rank (discovRE)	Multi-MH	Multi-k-MH	discovRE	rank (discovRE)	Multi-MH	Multi-k-MH	discovRE
ARM → DD-WRT	1;2	$1.3 \cdot 10^5$ ms	$4.3 \cdot 10^5$	43.8 ms	1	$2.2 \cdot 10^5$	$7.2 \cdot 10^5$	55.2 ms
ARM → Android	1;2	$5.7 \cdot 10^5$ ms	$1.9 \cdot 10^6$	49.5 ms	1	$9.7 \cdot 10^5$	$3.2 \cdot 10^6$	76.1 ms
ARM → ReadyNAS	1;2	$1.1 \cdot 10^6$ ms	$3.8 \cdot 10^6$	66.5 ms	1	$1.9 \cdot 10^6$	$6.4 \cdot 10^6$	80.9 ms
MIPS → DD-WRT	1;2	see above		47.2 ms	1	see above		54.7 ms
MIPS → Android	1;2			55.2 ms	1			72.4 ms
MIPS → ReadyNAS	1;2			65.7 ms	1			84.2 ms
x86 → DD-WRT	1;4	see above		43.0 ms	1	see above		51.2 ms
x86 → Android	1;2			58.7 ms	1			77.0 ms
x86 → ReadyNAS	1;5			69.8 ms	1			81.3 ms

TABLE VII: Results for searching different bugs in whole firmware images.

robust against code modifications. BINJUICE [40] normalized instructions of a basic block to extract its semantic "juice", which presents the relationships established by the block. Semantically similar basic blocks were then identified through lexical comparisons of juices. This approach only works at the basic block level and was extended to find similar code fragments that span several blocks. BINHASH [36] models functions as a set of features that represent the input-output behavior of a basic block. EXPOSÉ [52] is a search engine for binary code that uses simple features such as the number of functions to identify a set of candidate function matches. These candidates are then verified by symbolically executing both functions and leveraging a theorem prover. EXPOSÉ assumes that all functions use the `cdecl` calling convention, which is a very limiting assumptions even for binaries of the same architecture. David et al. [21] proposed to decompose functions into continuous, short, partial traces of an execution called *tracelets*. The similarity between two tracelets is computed by measuring how many rewrites are required to reach one tracelet from another. In the experiments the authors only considered functions with at least 100 basic blocks, which is rarely the case. Moreover, this method is not robust against compiler optimizations. TEDEM [54] automatically identifies binary code regions that are similar to code regions containing a known bug. It uses tree edit distances as a basic block centric metric for code similarity.

The most recent and advanced method to search for known bugs in binary code across different architectures was proposed by Pewny et al. [53]. First, the binary code is translated into the Valgrind intermediate representation VEX [9]. Then, concrete inputs are sampled to observe the input-output behavior of basic blocks, which grasps their semantics. Finally, the I/O behavior is used to find code parts that behave similarly to the bug signature. While the use of semantics similarity delivers precise results, it is too slow to be applicable to large code bases.

Previous work also explored methods that rely on dynamic analysis. Egele et al. [23] proposed Blanket Execution (BLEX) to match functions in binaries. To this end, BLEX executes each function for several calling contexts and collects the side effects of functions during execution under a controlled randomized environment. Two functions are considered similar if their corresponding side effects are similar. Dynamic analysis approaches use architecture-specific tools to run and instrument executables. For this reason, they are inherently difficult to extend to different architectures.

VII. CONCLUSION AND FUTURE WORK

We presented a system to efficiently identify already known bugs in binary code across different compilers, compiler optimizations, CPU architectures, and operating systems. In the preparation phase, a code base of known functions is analyzed and their numeric and structural features are stored. When a new bug is published, the vulnerable function is identified and that function is queried. Our approach employs a two-staged filtering approach to quickly identify the buggy function even in large code bases. The first filter relies on a set of robust numeric features to retrieve very similar functions based on the kNN algorithm. These functions serve as candidates to the next

stage that measures the structural similarity of these candidates to the target function. To this end we use an approximation of the maximum common subgraph isomorphism.

We implemented our methods in a tool called discovRE and evaluated its efficacy on real-world firmware images with up to 3 million basic blocks. Here, the preparation time was about 50 minutes. discovRE could correctly identify buggy function (e.g., Heartbleed or POODLE) from ARM, MIPS and x86 in three complete firmware images in about 80 ms.

A. Future Work

One outstanding source of a large distance between two functions with the same label is the different implementation of string operations. While some compilers invoke a function by pointing to the appropriate code, others apply the respective string operation directly. Additionally, some string operations are expressed differently, e.g., through *rep* prefixes vs. loops. This leads to substantial differences in the respective CFGs. Hence, a different representation could eliminate some of the described drawbacks of function inlining. For example, Yakdan et al. present a method to outline functions such as `strcpy` [61]. Other graph representations of functions are also interesting subjects for scrutinization and left for future work, e.g., the abstract syntax tree or the program dependence graph [62]. Additionally, we want to identify unrolled loops to further improve the recognition rate.

In the future, we plan to evaluate the false positive rate of discovRE. In our evaluation, the target function was in the queried code base. If the queried function does not exist in the binary or the binary has a patched version of the queried function, the result is false positive. We plan to investigate the possibility of introducing a threshold on the distance measure to classify functions as real matches or false positives. Additionally, the structural distance function can be modified into a similarity score between the queried function and the returned match.

In the future, we plan further research on using discovRE to assist manual reverse engineering of malware. In this setting, a malware analyst can use discovRE to find similar functions in a new version of malware. This helps her to quickly identify new functionality in the new version. Another line of research that we intend to follow in the future is employing discovRE for clustering malware families.

Acknowledgements

We thank our shepherd Thorsten Holz for his support in finalizing this paper. We would also like to thank the anonymous reviewers for their valuable feedback. We are grateful to Jannik Pewny for sharing the binaries and experiment results.

References

[1] BitDHT. http://bitdht.sourceforge.net/ (last visit: 2015-07-30).
[2] GnuPG. http://www.gnupg.org/ (last visit: 2015-07-30).
[3] SQLite. http://www.sqlite.org (last visit: 2015-01-30).
[4] stunnel. http://www.stunnel.org/ (last visit: 2015-07-30).
[5] The Heartbleed Vulnerability. http://heartbleed.com/. (last visit 2015-08-10).
[6] The IDA Pro Disassembler and Debugger. http://www.hex-rays.com/idapro/.
[7] The POODLE Vulnerability. https://www.poodletest.com/. (last visit 2015-08-10).
[8] The Shellshock Vulnerability. https://shellshocker.net/. (last visit 2015-08-10).
[9] Valgrind Documentation. http://valgrind.org/docs/manual/index.html. (last visit 2015-10-08).
[10] DD-WRT Firmware Image r21676, 2013. ftp://ftp.dd-wrt.com/others/eko/BrainSlayer-V24-preSP2/2013/05-27-2013-r21676/senao-eoc5610/linux.bin (last visit: 2015-04-14).
[11] ReadyNAS Firmware Image v6.1.6, 2013. http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip (last visit: 2015-04-14).
[12] A. V. Aho. *Compilers: Principles, Techniques and Tools, 2/e*. Pearson Education India, 2003.
[13] A. V. Aho, J. E. Hopcroft, and J. D. Vilman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1st edition, 1974.
[14] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium (NDSS)*, 2011.
[15] G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Science Department, University of Wisconsin-Madison, 2007.
[16] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
[17] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the 14th Network and Distributed System Security Symposium (NDSS)*, 2007.
[18] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3–4):255–259, 1998.
[19] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012.
[20] K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.
[21] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 37. ACM, 2014.
[22] T. Dullien and R. Rolles. Graph-based comparison of executable objects. *SSTIC*, 5:1–3, 2005.
[23] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
[24] H. Flake. Structural Comparison of Executable Objects. In *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 161–173, 2004.
[25] Free Software Foundation. GCC, the GNU Compiler Collection, 2010. http://gcc.gnu.org/ (last visit: 2015-01-30).
[26] D. Gao, M. K. Reiter, , and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.
[27] F. Gauthier, T. Lavoie, and E. Merlo. Uncovering Access Control Weaknesses and Flaws with Security-discordant Software Clones. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, 2013.
[28] I. Guilfanov. Fast Library Identification and Recognition Technology in IDA Pro, 12 1997. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml (last visit: 2015-01-22).
[29] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.
[30] ImageMagick Studio LLC. ImageMagick. http://www.imagemagick.org (last visit: 2015-07-30).
[31] Intel Corporation. Intel C++ Compiler. http://software.intel.com/en-us/intel-compilers/ (last visit: 2015-01-30).
[32] Intel Corporation. The OpenCV Library. http://sourceforge.net/projects/opencvlibrary/ (last visit: 2015-07-30).
[33] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals 1–3*, 2009.
[34] J. Jang, D. Brumley, and A. Agrawal. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[35] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

[36] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*, volume 1, pages 386–391. IEEE, 2012.

[37] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, 2006.

[38] 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):654–670, July 2002.

[39] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[40] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2013.

[41] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[42] B. Lee, C. Song, T. Kim, and W. Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[43] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004.

[44] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[45] R. Marimont and M. Shapiro. Nearest neighbour searches and the curse of dimensionality. *IMA Journal of Applied Mathematics*, 24(1):59–70, 1979.

[46] McCabe Software. More Complex = Less Secure. Miss a Test Path and You Could Get Hacked, 2012. http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf (last visit: 2015-04-14).

[47] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[48] Microsoft Corporation. Microsoft Visual C++, 2007. http://msdn.microsoft.com/visualc (last visit: 2015-01-30).

[49] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary Hunting with Interprocedural Control Flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, 2013.

[50] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2, 2009.

[51] G. Myles and C. Collberg. K-gram Based Software Birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*, 2005.

[52] B. H. Ng and A. Prakash. Exposé: Discovering Potential Binary Code Re-use. In *Proceedings of the 37th Annual IEEE Computer Software and Applications Conference (COMPSAC)*, 2013.

[53] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[54] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[55] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[56] Source Forge. http://sourceforge.net (last visit: 2015-02-20).

[57] The LAME Project. LAME MP3 Encoder. http://lame.sourceforge.net (last visit: 2015-01-30).

[58] M. Wall. Galib: A c++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 87:54, 1996.

[59] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[60] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[61] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[62] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[63] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[64] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

[65] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[66] M. Zitser, R. Lippmann, and T. Leek. Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT FSE)*, 2004.