LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics

Xiaoning Du*¶, Bihuan Chen†¶, Yuekang Li*, Jianmin Guo‡, Yaqin Zhou*, Yang Liu*, Yu Jiang‡
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
‡KLISS, BNRist, School of Software, Tsinghua University, China
§College of Information Science, Zhejiang Sci-Tech University, China
¶Co-First Authors

***Abstract*—Identifying potentially vulnerable locations in a code base is critical as a pre-step for effective vulnerability assessment; i.e., it can greatly help security experts put their time and effort to where it is needed most. Metric-based and pattern-based methods have been presented for identifying vulnerable code. The former relies on machine learning and cannot work well due to the severe imbalance between non-vulnerable and vulnerable code or lack of features to characterize vulnerabilities. The latter needs the prior knowledge of known vulnerabilities and can only identify similar but not new types of vulnerabilities.**

In this paper, we propose and implement a generic, lightweight and extensible framework, LEOPARD, to identify potentially vulnerable functions through program metrics. LEOPARD requires no prior knowledge about known vulnerabilities. It has two steps by combining two sets of systematically derived metrics. First, it uses complexity metrics to group the functions in a target application into a set of bins. Then, it uses vulnerability metrics to rank the functions in each bin and identifies the top ones as potentially vulnerable. Our experimental results on 11 real-world projects have demonstrated that, LEOPARD can cover 74.0% of vulnerable functions by identifying 20% of functions as vulnerable and outperform machine learning-based and static analysis-based techniques. We further propose three applications of LEOPARD for manual code review and fuzzing, through which we discovered 22 new bugs in real applications like `PHP`, `radare2` and `FFmpeg`, and eight of them are new vulnerabilities.

***Index Terms*—Program Metric, Vulnerability, Fuzzing**

I. INTRODUCTION

Vulnerabilities are one of the key threats to software security [42]. Security experts usually leverage guided fuzzing (e.g., [14, 50, 66, 67]), symbolic execution (e.g., [12, 17, 27, 60]) or manual auditing to hunt vulnerabilities. As only a few vulnerabilities are scattered across a large code base, vulnerability hunting is a very challenging task that requires intensive knowledge and is comparable to finding "a needle in a haystack" [81]. Therefore, a large amount of time and effort is wasted in analyzing the non-vulnerable code. In that sense, identifying potentially vulnerable code in a code base can guide vulnerability hunting and assessment in a promising direction.

There are two types of existing techniques to automatically identify vulnerabilities: metric-based and pattern-based techniques. Metric-based techniques, inspired by bug prediction [16, 28, 30, 38, 46, 49, 78], leverage supervised or unsupervised machine learning to predict vulnerable code mostly at the granularity level of a source file. Following security experts' belief that complexity is the enemy of software security [40], they use complexity metrics [21, 44, 45, 55, 56] as features, or combine them with code churn metrics [26, 54, 58], token frequency metrics [31, 52, 65, 79], dependency metrics [43, 47, 48, 81], developer activity metrics [54, 58] and execution complexity metrics [57]. On the other hand, pattern-based techniques leverage patterns of known vulnerabilities to identify potentially vulnerable code through static analysis. The patterns are formulated based on the syntax or semantics abstraction of a certain type of vulnerabilities, e.g., missing security checks on security-critical objects [59, 74], security properties [63], code structures [72], and vulnerability specifications [37, 71].

While vulnerability identification has been attracting great attention, some problems still remain. On one hand, metric-based techniques are mostly designed for one single application (or a few applications of the same type). Thus, they might not work on a variety of diverse applications as machine learning may over-fit to noise features. Moreover, while an empirical connection between vulnerabilities and bugs exist, the connection is considerably weak due to the differences between vulnerabilities and bugs [15]. As a result, the research on bug prediction cannot directly translate to vulnerability identification. Unfortunately, the existing metric-based techniques use the similar metrics as those in bug prediction, and thus fail to investigate the characteristics of vulnerabilities.

On the other hand, metric-based and pattern-based techniques mostly require a great deal of prior knowledge about vulnerabilities. In particular, a large number of known vulnerabilities are needed for effective supervised machine learning in some metric-based techniques. The number of vulnerabilities is much smaller than the number of bugs, and the imbalance between non-vulnerable and vulnerable code is severe, which hinders the applicability of supervised machine learning to vulnerable code identification. Similarly, a prerequisite of those pattern-based techniques is the existence of known vulnerabilities as the guideline to formulate patterns. They can only identify similar but not new vulnerabilities. Further, patterns are often application-specific, and thus those techniques are usually used as in-project but not cross-project vulnerable code identification.

In this paper, we propose a vulnerability identification framework, named LEOPARD[1], to identify potentially vulnerable functions in C/C++ applications. LEOPARD is designed to be *generic* to work for different types of applications, *lightweight* to support the analysis of large-scale applications and *extensible* with domain-specific data to improve the accuracy. We design LEOPARD as a pre-step for vulnerability assessment, but not to directly pinpoint vulnerabilities. We propose three different applications of LEOPARD to guide security experts during the manual auditing or automatic fuzzing by narrowing down the space of potentially vulnerable functions.

LEOPARD does not require any prior knowledge about known vulnerabilities. It works in two steps by combining two sets of systematically derived program metrics, i.e., complexity metrics and vulnerability metrics. Complexity metrics capture the complexity of a function in two complementary dimensions: the cyclomatic complexity of the function, and the loop structures in the function. Vulnerability metrics reflect the vulnerable characteristics of functions in three dimensions: the dependency of the function, pointer usage in the function, and the dependency among control structures within the function.

LEOPARD first uses complexity metrics to group the functions in a target application into a set of bins. Then, LEOPARD leverages vulnerability metrics to rank the functions in each bin and identify the top functions in each bin as potentially vulnerable. We propose such a binning-and-ranking approach as there often exists a proportional relation between complexity and vulnerability metrics, which is evidenced in our experimental study. As a result, each bin has a different level of complexity, and our framework can identify vulnerabilities at all levels of complexity without missing low-complexity ones.

We implemented the proposed framework to obtain complexity and vulnerability metrics for C/C++ programs. We evaluated the effectiveness and scalability of our framework with 11 different types of real-world projects. LEOPARD can cover 74.0% of vulnerable functions by identifying 20% of functions as potentially vulnerable, outperforming both typical machine learning-based and static analysis-based techniques. Applying LEOPARD on `PHP`, `MJS`, `XED`, `FFmpeg` and `Radare2` and with further manual auditing or automatic fuzzing, we discovered 22 new bugs, among which eight are new vulnerabilities.

In summary, our work makes the following contributions.

- We propose a generic, lightweight and extensible framework to identify potentially vulnerable functions, which incorporates two sets of program metrics.
- We propose three different applications of LEOPARD to guide security experts during the manual auditing or automatic fuzzing to hunt for vulnerabilities.
- We implemented our framework and conducted large-scale experiments on 11 real-world projects to demonstrate the effectiveness and scalability of our framework.
- We demonstrated three application scenarios of our framework and found 22 new bugs.

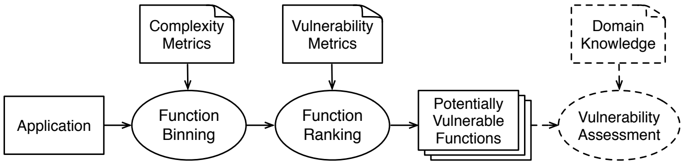


Fig. 1: An Overview of the Proposed Framework

II. METHODOLOGY

In this section, we present the overview of LEOPARD and elaborate each step of the proposed approach.

A. Overview

Fig. 1 presents the work flow of LEOPARD, which is designed to be generic, lightweight and extensible. The input is the source code of a C/C++ application. LEOPARD works in two steps: function binning and function ranking, and returns a list of potentially vulnerable functions for vulnerability assessment.

In the first step (§ II-B), we use complexity metrics to group all functions in the target application into a set of bins. The complexity metrics capture the complexity of a function in two dimensions: the function itself (i.e., cyclomatic complexity) and the loop structures in the function (e.g., the number of nested loops). Each bin has a different level of complexity, which is designed to identify vulnerabilities at all levels of complexity (i.e., avoid missing vulnerable functions with low-complexity).

In the second step (§ II-C), we use vulnerability metrics to rank the functions in each bin in order to identify the top functions in each bin as potentially vulnerable. The vulnerability metrics capture the vulnerable characteristics of a function in three dimensions: the dependency of the function (e.g., the number of parameters), the pointer usage in a function (e.g., the number of pointer arithmetic) and the dependency of control structures in the function (e.g., the number of nested control structures). By incorporating such metrics, we can have a high potential of characterizing and identifying vulnerable functions.

LEOPARD is designed to support and facilitate confirmative vulnerability assessments, e.g., to guide security experts during automatic fuzzing [14, 50, 66, 67] or manual auditing by providing potentially vulnerable function list and the corresponding metrics information. With such knowledge, security experts can prioritize the assessment order, choose the appropriate analysis technique, and analyze the root cause. Further, based on application-specific domain knowledge (e.g., vulnerability history and heavily fuzzed function lists), security experts can further rank or filter the potentially vulnerable functions to focus on those more interesting functions.

Using program metrics in a simple binning-and-ranking way makes LEOPARD satisfy our design principle of being generic and lightweight. It is applicable to any large-scale applications of any type and does not require prior knowledge about known vulnerabilities. The two sets of metrics are comprehensive, but also are extensible with new metrics as we gather more usage feedback from security experts (see discussion in § V). Thus, LEOPARD also satisfies our design principle of being extensible such that it can be further enhanced.

[1] Leopard is known for its opportunistic hunting behavior, broad diet, and strength, which reflect the identification capabilities we are pursuing.

TABLE I: Complexity Metrics of a Function

Dimension	ID	Metric Description
CD1: Function	C1	Cyclomatic complexity
CD2: Loop Structures	C2	# of loops
	C3	# of nested loops
	C4	Maximum nesting level of loops

B. Function Binning

Different vulnerabilities often have different levels of complexity. To identify vulnerabilities at all levels of complexity, in the first step, we categorize all functions in the target application into a set of bins based on complexity metrics. As a result, each bin represents a different level of complexity. Afterwards, the second step (§ II-C) plays the prediction role via ranking. Such a binning-and-ranking approach is designed to avoid missing low-complexity vulnerable functions.

Complexity Metrics. By "complexity", we refer to the approximate number of paths in a function, and derive the complexity metrics of a function from its structural complexity. A function often has loop and control structures, which are the main sources of structural complexity. Cyclomatic complexity [39] is a widely-used metric to measure the complexity, but without reflection of the loop structures. Based on such understanding, we introduce the complexity of a function with respect to these two complementary dimensions, as shown in Table I.

Function metric (C1) captures the standard cyclomatic complexity [39] of a function, i.e., the number of linearly independent paths through a function. A higher value of C1 means that the function is likely more difficult to analyze or test.

Loop structure metrics (C2–C4) reflect the complexity resulting from loops, which can drastically increase the number of paths in the function. Metrics include the number of loops, the number of nested loops, and the maximum nesting level of loops. Loops are challenging in program analysis [68] and hinder vulnerability analysis. Basically, the higher these metrics, the more (and possibly longer) paths need to be considered and the more difficult to analyze the function.

Binning Strategy. Given the values of these complexity metrics for functions in the target application, we compute a *complexity score* for each function by adding up all the complexity metric values, and then group the functions with the same score into the same bin. Here we do not use a range-based binning strategy (i.e., grouping the functions whose scores fall into the same range into the same bin) as it is hard to determine the suitable granularity of the range. Such a simple strategy not only makes our framework lightweight, but also works well, as evidenced by our experimental study in § IV-C.

C. Function Ranking

Different from the structural complexity metrics, in the second step, we derive a new set of vulnerability metrics according to the characteristics of general causes of vulnerabilities, and then rank the functions and identify the top ones in each bin as potentially vulnerable based on the vulnerability metrics. Existing metric-based techniques [44, 45] rarely employ any vulnerability-oriented metrics, and make no differentiation between complexity metrics and vulnerability metrics. Here, we propose and incorporate vulnerability metrics to have a high potential of characterizing and identifying vulnerable functions.

TABLE II: Vulnerability Metrics of a Function

Dimension	ID	Metric Description
VD1: Dependency	V1	# of parameter variables
	V2	# of variables as parameters for callee function
VD2: Pointers	V3	# of pointer arithmetic
	V4	# of variables involved in pointer arithmetic
	V5	Max pointer arithmetic a variable is involved in
VD3: Control Structures	V6	# of nested control structures
	V7	Maximum nesting level of control structures
	V8	Maximum of control-dependent control structures
	V9	Maximum of data-dependent control structures
	V10	# of if structures without else
	V11	# of variables involved in control predicates

Vulnerability Metrics. Most critical types of vulnerabilities in C/C++ programs are directly or indirectly caused by memory management errors [61] and/or missing checks on some sensitive variables [74] (e.g., pointers). Resulting vulnerabilities include but are not limited to memory errors, access control errors (e.g., missing checks on user permission), and information leakage. Actually, the root causes of many denial of service and code execution vulnerabilities can also be traced back to these causes. The above mentioned types account for more than 70% of all vulnerabilities [11]. Hence, it is possible to define a set of vulnerability metrics that are compatible with major vulnerability types. Here we would not favor any specific types of vulnerabilities, e.g., to include metrics like division operation which is closely related to divide-by-zero, while the exploration of type-specific metrics is worth of investigation in the future. With either high or low complexity scores, vulnerable functions we focus on are mainly with complicated and compact computations, which are independent from the number of paths in the function. Based on these observations, we introduce the vulnerability metrics of a function w.r.t. three dimensions, as summarized in Table II.

Dependency metrics (V1–V2) characterize the dependency relationship of a function with other functions, i.e., the number of parameter variables of the function and the number of variables prepared by the function as parameters of function calls. The more dependent with other functions, the more difficult to track the interaction.

Pointer metrics (V3–V5) capture the manipulation of pointers, i.e., the number of pointer arithmetic, the number of variables used in pointer arithmetic, and the maximum number of pointer arithmetic a variable is involved in. Member access operations (e.g., ptr→m), deference operations (e.g., *ptr), incrementing pointers (e.g., ptr++), and decrementing pointers (e.g., prt--) are all pointer arithmetics. The number of pointer arithmetic can be obtained from the Abstract Syntax Tree (AST) of the function via simple counting. These operations are closely related to sensitive memory manipulations, which can increase the risk of memory management errors.

Alongside, we count how many unique variables are used in the pointer arithmetic operations. The more variables get involved, the more challenging for programmers to make correct decisions. For these variables, we also examine how many pointer arithmetic operations they are involved in and record the maximum value. Frequent operations on the same pointer

```c
1  void fibonacci(int *res, int n) {
2    if (n <= 0) {
3      return;
4    }
5    res[0] = 0;
6    res[1] = 1;
7    if (n > 1) {
8      if (n == 3) {
9        res[2] = 1;
10       return;
11     }
12     for(int i = 2; i <= n; i++) {
13       res[i] = res[i-1] + res[i-2];
14     }
15   }
16 }
```

Fig. 2: A Function to Calculate Fibonacci Series

make it harder to track its value and guarantee the correctness. In a word, the higher these metrics, the higher chance to cause complicated memory management problems, and thus higher chance to dereference null or out-of-bound pointers.

Control structure metrics (V6–V11) capture the vulnerability due to highly coupled and dependent control structures (such as *if* and *while*), i.e., the number of nested control structures pairs, the maximum nesting level of control structures, the maximum number of control structures that are control- or data-dependent, the number of *if* structures without explicit *else* statement, and the number of variables that are involved in the data-dependent control structures. We explain the above metrics with an example (Fig. 2) calculating Fibonacci series. There are two pairs of nested control structures, *if* at Line 7 respectively with *if* at Line 8 and *for* at Line 12. Obviously, the maximum nesting level is two, with the outer structure as *if* at Line 7. The maximum of control-dependent control structures is 3, including *if* at Line 7 and Line 8, and *for* at Line 12. The maximum of data-dependent control structures is four since conditions in all four control structures make checks on variable n. All three *if* statements are without *else*. There are two variables, i.e., n and i involved in the predicates of control structures. Actually, the more variables used in the predicates, the more likely to makes error on sanity checks. The higher these metrics, the harder for programmers to follow, and the more difficult to reach the deeper part of the function during vulnerability hunting. Stand-alone *if* structures are suspicious for missing checks on the implicit *else* branches.

There usually exists a proportional relation between complexity and vulnerability metrics, because the more complex the (independent path and loop) structures of a function, the higher chance the variables, pointers and coupled control structures are involved. The complexity metrics are used to approximate the number of paths in the function, which are neutral to the vulnerable characteristics. Importantly, for the set of control structure metrics used as vulnerability indicators, they describe a different aspect of properties than complexity metrics. First, whether control structures are nested or dependent, or whether *if* are followed by *else*, are independent to cyclomatic complexity metrics. Second, intensively coupled control structures are good evidence of vulnerability. Instead of directly ranking functions with complexity and/or vulnerability metrics, we propose a binning-and-ranking approach to avoid missing less complicated but vulnerable functions, as will be evidenced in § IV-B.

Ranking Strategy. Based on the values of these metrics for the functions, we compute a *vulnerability score* for each function by adding up all the metric values, rank the functions in each bin according to the scores, and cumulatively identify the top functions with highest scores in each bin as potential vulnerable functions. During the selection, we identify the top k functions from each bin where k is initially 1, and increase by 1 in each selection iteration. Notice that we may take more than k functions as we treat functions with the same score equally. This selection stops when an appropriate portion (i.e., p) of functions has been selected. Here p can be set by users. Similar to the binning strategy, we adopt a simple ranking strategy to make our framework both lightweight and effective.

III. Applications of Leopard

Leopard is not designed to directly pinpoint vulnerabilities but to assist confirmative vulnerability assessment. Leopard outputs a list of potential vulnerable functions with complexity metrics and vulnerability metrics scores, which can provide useful insight for further vulnerability hunting. In this section, we demonstrate three different ways to apply the results generated by Leopard for finding vulnerabilities. With Leopard, we found 22 new bugs in five widely-used real-world programs. The detailed experimental results will be introduced in § IV-F.

Manual Auditing. In general, with the help of Leopard, manual auditing (e.g., code review) can be greatly improved w.r.t. effectiveness and efficiency. Instead of auditing all the functions [22], security experts can focus on only those potentially vulnerable functions that are identified by Leopard.

Furthermore, the vulnerability metrics produced by Leopard may help security experts to quickly identify the root cause of vulnerabilities with their domain knowledge, especially for complicated large functions. For example, if a vulnerable function has a large number of instances of *if-without-else*, security experts could pay attention to the logic of the missing *else* to see if there are potential missing checks; and if a vulnerable function has a large number of pointers, security experts could focus on the memory allocation and deallocation operations to see if there are potential dangling pointers. Although these metrics cannot directly pinpoint the root cause, it can provide explicit hints on the possible root cause.

Target Identification for Directed Fuzzing. Fuzzing has been shown as an effective testing technique to find vulnerabilities. Specifically, greybox fuzzers (e.g., AFL [4] and its variants [13, 14]) have gained the popularity and been proven to be practical for finding vulnerabilities in real-world applications.

Current greybox fuzzers aim to cover as many program states as possible within a given time budget. However, higher coverage does not necessarily imply finding more vulnerabilities because fuzzers are blindly exploring all possible program states without focusing the efforts on the more vulnerable functions. Recently, directed greybox fuzzers (e.g., AFLGo [13] and Hawkeye [20]) are proposed to guide the fuzzing execution towards a predefined vulnerable function (a.k.a. target site) to either reproduce the vulnerability or check whether a patched function is still vulnerable [13].

Since LEOPARD produces a list potential vulnerable functions, a straightforward application with directed greybox fuzzers is to set potential vulnerable functions as target sites. In this way, we can quickly confirm whether a potentially vulnerable function is really vulnerable or a false positive by directing the fuzzer to concentrate on the function. Note that although the fuzzer can reach a vulnerable function, the vulnerability hidden in the function may not always be triggered. But still, directed fuzzing has been shown as an effective technique to reproduce vulnerabilities [13]. To demonstrate the idea, we utilize a directed fuzzing tool, Hawkeye [20], which is built upon an extensible fuzzing framework FOT [19] and reported to outperform ALFGo [13]. However, due to the large number of the potential vulnerable functions generated by LEOPARD, it is ineffective to set all potential vulnerable functions as target sites as it may confuse the fuzzer where to guide. To this end, we choose to separate the target application into smaller modules based on its architecture design or simply namespace, and then let the Hawkeye to fuzz with the targets grouped by modules separately.

Seed Prioritization for Fuzzing. Greybox fuzzers often keep interesting test inputs (i.e., seeds) for further fuzzing. These seeds need to be continuously evaluated to decide which of them should be prioritized. By default, most fuzzers (e.g., AFL) prefer seeds with "smaller file size" and "shorter execution time" or "more edge (basic-block transition) coverage", which are not vulnerability-aware decisions.

Since LEOPARD assigns each function a vulnerability score and a complexity score, we can use these scores to help to evaluate which seed should be prioritized such that the fuzzer can find more vulnerabilities in the given time budget. For this purpose, we extended *FOT* by enabling it to accept external function-level scores for seed prioritization. The detailed seed evaluation process is explained as follows. First, we calculate a priority score for each function based on the binning-and-ranking strategy. For a function $\mathcal{F}$ within top k, its priority score is calculated using the following formula:

$$priority\_score(\mathcal{F}) = 100 - \frac{\sum_{i=1}^{k} \mathcal{N}_i}{\mathcal{N}} \cdot 100 \qquad (1)$$

where $\mathcal{N}_i$ is the number of functions with rank i and $\mathcal{N}$ is the total number of all functions. For example, if the top 1 functions contribute a portion of 20% to the total number of all functions, then these functions are assigned with a score of 80 $(100-20)$. Then, the function-score mapping is provided to *FOT*. After executing a test input (i.e., seed), the fuzzer can get an execution trace consist of functions. Then the fuzzer will accumulate the priority scores of the functions on the execution trace to form the priority score of that trace. As a result, each seed is associated with a trace priority score representing its *vulnerableness*. When the fuzzer chooses the next seed to fuzz, it will select the one with highest trace priority score.

IV. EVALUATION

LEOPARD is implemented in 11K lines of Python code. Specifically, we used Joern [71] to extract the values of complexity and vulnerability metrics, given the source code of an application. More details of the implementation and evaluation are available at our website [6].

TABLE III: Details of the Target Applications

Project	SLOC	#Func.	Vul. Func.	CVE	Excl. CVE
BIND 9.11.0	504K	9,462	9	3	3
Binutils 2.28	3,336K	24,713	84	37	24
FFmpeg 3.1.3	986K	19,336	38	26	6
FreeType 2.3.9	126K	1,847	74	48	18
Libav 11.8	583k	10,583	8	6	8
LibTIFF 4.0.6	118K	1,394	20	12	24
libxslt 1.1.28	47K	666	5	3	1
Linux 4.12.8	17,103K	488,960	256	104	32
OpenSSL 1.0.1t	360K	6,649	42	17	3
SQLite 3.8.2	172K	3,651	10	7	2
Wireshark 2.2.0	3,551K	33,564	152	74	31
Total	26,886K	600,825	698	337	152

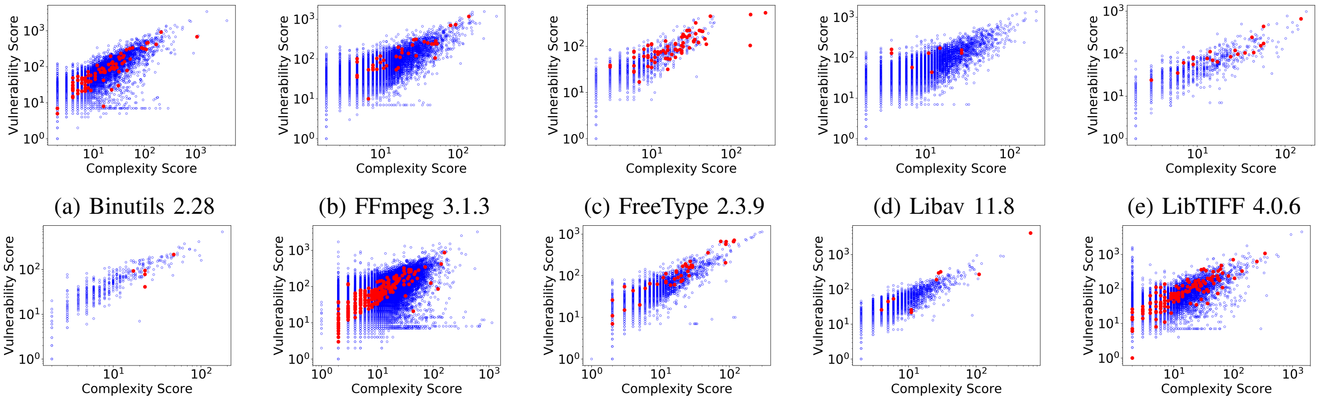
A. Evaluation Setup

Target Applications. We used 11 real-world open-source projects that represent a diverse set of applications. `BIND` is the most widely used Domain Name System (DNS) software. `Binutils` is a collection of binary tools. `FFmpeg` is the leading multimedia framework. `FreeType` is a library to render fonts. `Libav` is a library for handling multimedia data, which was originally forked from `FFmpeg`. `LibTIFF` is a library for reading and writing Tagged Image File Format (TIFF) files. `libxslt` is the XSLT C library for the GNOME project. `Linux` is a monolithic Unix-like computer operating system kernel. `OpenSSL` is a robust and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. `SQLite` is a relational database management system. `Wireshark` is a network traffic analyzer for Unix and Unix-like operating systems.

The details of each target application are reported in Table III. The first column gives the project version, the second column reports the source lines of code, and the third column lists the total number of functions in each project. The last three columns report the number of vulnerable functions, CVEs (Common Vulnerabilities and Exposures), and CVEs excluded from our research, collected as ground truth (see below). Here, we chose the recent versions of the projects that had large number of CVEs. The number of functions ranges from 666 for `libxslt` to 488,960 for `Linux`, which is diverse enough to show the generality of our framework. In total, 26,886K lines of code and 600,825 functions are studied, which makes our study large-scale and its results reliable.

Ground Truth. To obtain the ground truth for evaluating the effectiveness of LEOPARD, we first manually identified the list of vulnerabilities that were disclosed before July 2018 in the 11 projects from two vulnerability database websites: CVE Details [11] and National Vulnerability Database [7], i.e., we collected all the vulnerabilities reported for the given version of the project from its release date to July 2018. CVEs in external libraries used in a project are not claimed to the project.

The full list of CVEs in most projects are recorded by the above two websites. However, the patches of the CVEs are not well maintained and difficult to collect. We obtained available patches of these CVEs in the 11 projects from an industrial



(a) Binutils 2.28 (b) FFmpeg 3.1.3 (c) FreeType 2.3.9 (d) Libav 11.8 (e) LibTIFF 4.0.6

(f) libxslt 1.1.28 (g) Linux 4.12.8 (h) OpenSSL 1.0.1t (i) SQLite 3.8.2 (j) Wireshark 2.2.0

Fig. 3: Vulnerability Score vs. Complexity Score for Non-Vulnerable Functions (in Blue) and Vulnerable Functions (in Red)

collaborator, who offers vulnerability scanning services for C/C++ programs. Functions that are patched to fix the vulnerability are identified as vulnerable. The results are reported in the fourth and fifth columns of Table III. As an example, we display the CVE list, available patches and corresponding patched functions of `Libav` at our website [6].

Some CVEs failed to be included in our research, as shown in the last column of Table III because (i) there is no public detail about the fix that can directly identify the affected vulnerable functions as either the CVE affects some closed source projects or other reasons (e.g., CVE-2015-6607 and CVE-2015-5895 for `SQLite 3.8.2`); (ii) the fix does not involve direct code change on functions (e.g., CVE-2016-7958 for `Wireshark 2.2.0` and CVE-2016-2183 for `OpenSSL 1.0.1t`).

Research Questions. We designed the experiments to answer the following research questions:

- **Q1.** Is the binning step before the ranking step reasonable? (§ IV-B)
- **Q2.** Is our binning-and-ranking approach effective, and can it outperform baseline approaches, machine learning-based techniques and some off-the-shelf static scanners? (§ IV-C)
- **Q3.** What is the sensitivity of the metrics to the effectiveness of our framework? (§ IV-D)
- **Q4.** What is the performance overhead (i.e., scalability) of our framework? (§ IV-E)
- **Q5.** What are the potential application scenarios of LEOPARD? (§ IV-F)

B. *Rationality of Binning before Ranking (Q1)*

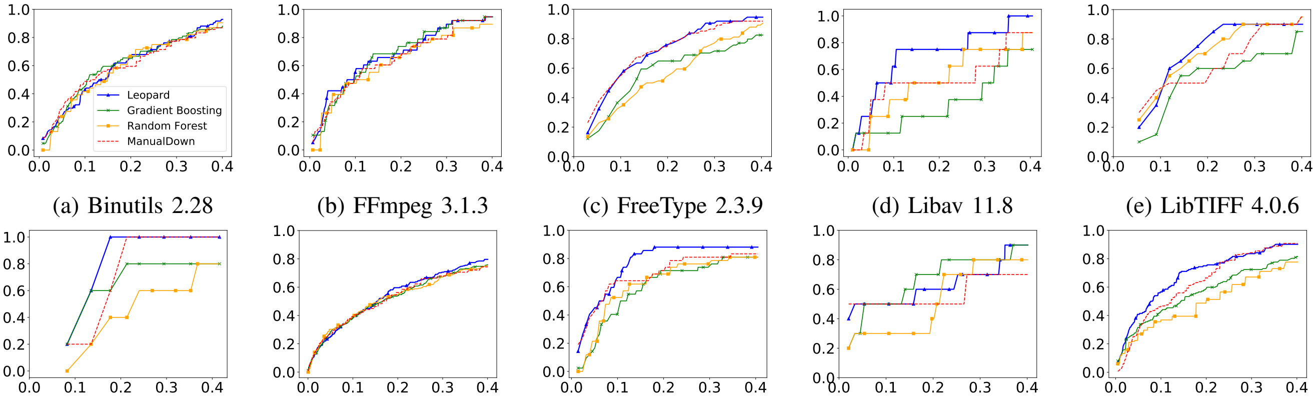
To answer this question, we first computed the complexity score and vulnerability score, as in § II-B and § II-C, for each function in all the projects (as shown in Table III). Then we plotted in Fig. 3 the relationship between complexity score (i.e., x-axis) and vulnerability score (i.e., y-axis) using logarithmic scale, where vulnerable and non-vulnerable functions were respectively highlighted in red and blue. The result of `BIND` is omitted for space limitations but is available on our website [6].

We can see from Fig. 3 that all projects share the similar patterns; vulnerable functions are scattered across non-vulnerable functions w.r.t. complexity score and vulnerability score; and there exists an approximately proportional relation between complexity score and vulnerability score for vulnerable functions. Therefore, if we directly ranked the functions based on complexity metrics and/or vulnerability metrics, we would always favor those functions with high complexity score and high vulnerability score, and miss those with low-complexity but vulnerable (e.g., vulnerable functions located in the first 3 bins in Fig. 3a, 3g and 3j). Instead, by first binning the functions according to complexity score and then ranking the functions in each bin according to vulnerability score, our framework can effectively identify the potentially vulnerable functions at all levels of complexity (see details in § IV-C). For all 11 projects, the number of bins ranges from 56 to 206 with an average of 114. Each bin has 301 functions on average, and 22% of bins contain vulnerable functions. Details of the function distribution among bins can be found at our website [6]. As can be seen from Fig. 3, bins with smaller complexity scores have more functions, and bins with larger complexity scores have more vulnerable functions. Sparsity of bins with larger complexity scores benefits the selection of most vulnerable functions, while our ranking in bins with smaller complexity scores gives more chance to identify less complex but vulnerable functions. Moreover, Fig. 3 also visually indicates the severe imbalance between non-vulnerable and vulnerable functions (see the third and fourth columns of Table III), which indicates traditional machine learning will over-fit and be less effective (more details will be discussed in § IV-C).

Our binning-and-ranking approach is reasonable for predicting vulnerable functions at all levels of complexity.

C. *Effectiveness of Binning-and-Ranking (Q2)*

We ran LEOPARD on all the projects; and analyzed its effectiveness when selecting different portion of functions, i.e., the parameter p in the ranking step (see § II-C). Here we used the percentage of functions (i.e., *Iden. Func.*) that are identified by LEOPARD as potentially vulnerable, and the percentage of vulnerable functions (i.e., *Cov. Vul. Func.*) that are covered by those identified potentially vulnerable functions as the two indicators of the effectiveness of our framework. These two indicators are used throughout the evaluation section.



(a) Binutils 2.28 (b) FFmpeg 3.1.3 (c) FreeType 2.3.9 (d) Libav 11.8 (e) LibTIFF 4.0.6

(f) libxslt 1.1.28 (g) Linux 4.12.8 (h) OpenSSL 1.0.1t (i) SQLite 3.8.2 (j) Wireshark 2.2.0

Fig. 4: Percentage of Functions (*Iden. Func.*) that are Identified as Potentially Vulnerable, and Percentage of Vulnerable Functions (*Cov. Vul. Func.*) that are Covered by Those Identified Potentially Vulnerable Functions

TABLE IV: Identified and Covered Vulnerable Functions

Project	Iden. Func. (%)					
	5%	10%	15%	**20%**	25%	30%
BIND 9.11.0	55.6	55.6	66.7	**66.7**	66.7	88.9
Binutils 2.28	23.8	42.9	51.2	**65.5**	71.4	78.6
FFmpeg 3.1.3	42.1	55.3	65.8	**68.4**	78.9	89.5
FreeType 2.3.9	16.2	52.7	63.5	**75.7**	83.8	90.5
Libav 11.8	25.0	62.5	75.0	**75.0**	75.0	87.5
LibTIFF 4.0.6	0.0	35.0	65.0	**80.0**	90.0	90.0
libxslt 1.1.28	0.0	20.0	60.0	**100.0**	100.0	100.0
Linux 4.12.8	24.9	38.9	48.3	**59.6**	64.9	70.6
OpenSSL 1.0.1t	42.9	66.7	83.3	**88.1**	88.1	88.1
SQLite 3.8.2	50.0	50.0	50.0	**60.0**	60.0	70.0
Wireshark 2.2.0	40.8	56.6	71.1	**75.0**	79.6	83.6
Average	29.2	48.7	63.6	**74.0**	78.0	85.2

TABLE V: Comparison of LEOPARD to Existing Approaches

Approach	Iden. Func. (%)					
	5%	10%	15%	20%	25%	30%
LEOPARD	29.2	**48.7**	**63.6**	**74.0**	**78.0**	**85.2**
ManualDown	**34.3**	47.9	54.4	63.7	70.6	78.2
Random Forest	25.8	37.7	48.8	58.8	68.7	75.6
Gradient Boosting	22.1	39.3	54.4	60.9	67.8	73.0

The results are shown in Fig. 4, where the x-axis denotes *Iden. Func.* and the y-axis denotes *Cov. Vul. Func.*. The legends are only shown in Fig. 4a and omitted in others for clarity; and the result of BIND is omitted but available on the website [6]. In general, as *Iden. Func.* increases, the indicator *Cov. Vul. Func.* also increases. For a small value (e.g., 20%) of *Iden. Func.*, our binning-and-ranking approach can achieve a high value for *Cov. Vul. Func.* (e.g., 74%). Furthermore, we also report how many vulnerable functions are covered when we identify certain percentage of functions as vulnerable in Table IV. When identifying 5 %, 10%, 15%, 20%, 25% and 30% of functions as vulnerable, we can cover 29%, 49%, 64% 74%, 78% and 85% of vulnerable functions. This means by identifying a small part of functions as vulnerable, we cover a large portion of vulnerable functions, which can narrow down the assessment space for security experts.

Comparison to Baseline Approaches. A recent study [80] on 42 existing cross-project defect prediction models and two state-of-the-art unsupervised defect prediction models [46, 78] has indicated that, simply ranking functions based on source lines of code (SLOC) in an increasing (i.e., ManualUp) or decreasing (i.e., ManualDown) order can achieve comparable or even superior prediction performance compared to most defect prediction models. We put the results of ManualUp (which is much worse than LEOPARD) at our website [6], and only show results of ManualDown in this section.

In Fig. 4, the comparison of *Cov. Vul. Func.* between LEOPARD and ManualDown is shown for each project. LEOPARD shows better results for all projects except for Binutils and FreeType, where both approaches have similar performance. On average, compared to ManualDown, 9.2%, 10.3% and 7.4% improvement are achieved when identifying 15%, 20% and 25% of functions as vulnerable, as shown in Table V; and we identify 15.6%, 13.8% and 11.8% less codes (measured in SLOC) than ManualDown. On average, 96.8% of ManualDown's true positives are covered by LEOPARD. This demonstrates the effectiveness of LEOPARD.

Comparison to Machine Learning-Based Techniques. We also conducted experiments to compare our framework with four machine learning-based techniques, namely random forest (RF), gradient boosting (GB), naive Bayes (NB) and support vector classification (SVC). The four techniques used all 4 complexity metrics and 11 vulnerability metrics as the features, and conducted a cross-project prediction by first training a model with the data from ten of the 11 projects and then using the model to predict the probability of being vulnerable for the functions in the other one project. By rotating the project to predict, we obtained the prediction results for all 11 projects. A larger predicted probability indicates that a function is more likely vulnerable. We rank the functions according to the probabilities, and identify a list of high-probability functions as vulnerable. A fair comparison to LEOPARD can be drawn when the same number of functions is identified. The results are shown in Fig. 4 and Table V.

As shown in Fig. 4, an obvious shortcoming of RF and GB is the unstable performance among different projects. It indicates that machine learning approaches highly depend on the large knowledge base of various vulnerable functions, which are however hard to obtain. Specifically, RF only shows similar or slightly better performance than LEOPARD in Fig. 3a and 3b, while GB only shows similar performance in Fig. 3a,

TABLE VI: Number of Alarms and Recall of Static Scanners

Project	S***		Cppcheck	
	#Alarm	Recall	#Alarm	Recall
BIND 9.11.0	250	0.0	45	0.0
Binutils 2.28	106	0.0	261	0.012
FFmpeg 3.1.3	42	0.0	306	0.0
FreeType 2.3.9	0	0.0	82	0.054
Libav 11.8	19	0.0	138	0.0
LibTIFF 4.0.6	76	0.1	10	0.0
libxslt 1.1.28	20	0.0	6	0.0
Linux 4.12.8	-	-	3864	0.0
OpenSSL 1.0.1t	76	0.0	33	0.0
SQLite 3.8.2	20	0.0	37	0.0
Wireshark 2.2.0	0	0.0	115	0.007

3b and 3i. LEOPARD outperforms RF and GB in Fig. 3c, 3d, 3e 3f, 3g, 3h and 3j. Both RF and GB performs even worse than the ManualDown baseline in Fig. 3c, 3h and 3j. As numerically shown in Table V, when identifying 20% of functions, RF and GB separately cover 15.2% and 13.1% less of ground truth than LEOPARD. Again, LEOPARD does not rely on any prior knowledge about a large set of vulnerabilities but machine learning-based techniques do. NB and SVC presented extremely lower recalls among the four typical machine learning algorithms. Hence, we omitted the results and put them at our website [6]. Note that 11 projects may not be an adequate dataset for training and testing, especially given the severe imbalance between vulnerable and non-vulnerable functions, the validity of conclusions drawn can be threatened. However, such a prerequisite for prior knowledge of vulnerable functions motivate our design of LEOPARD.

Comparison to Static Scanners. We also applied two popular static software scanner tools to investigate their vulnerability prediction capability on our dataset, including an open source tool, Cppcheck [10], and a commercial tool. To avoid legal disputes, we hide the name of the commercial one and refer it as S***. Cppcheck and S*** are among the most popular static code analysis tools used to detect bugs and vulnerabilities in software. Both tools report the suspicious vulnerable statements. Whenever an alarm locates within the vulnerable functions in our ground truth, we claim a true positive for that tool. The number of total alarms reported by these two tools and the recall can be found in Table VI. Cppcheck was able to analyze all 11 projects and identified a few vulnerable functions in `Binutils`, `FreeType` and `Wireshark`. S*** failed to analyze `Linux`; and for the other 10 projects only a few vulnerable functions are detected in `LibTIFF`. Static scanners often rely on very concrete vulnerability patterns. Subtle pattern mismatch will cause false positives and negatives. Thus. their recalls are nearly 0, which indicate that they are not promising for general vulnerability identification.

False Negative Analysis. By examining the vulnerable functions that LEOPARD fails to cover when 40% functions are identified, we summarize three main reasons for false negatives: 1) they are involved in some logical vulnerabilities which are hard to be revealed by function metrics; 2) they are implicated via some fixes indirectly related to the CVE, e.g., if a fix changes the function signature, callers of this function should not be counted as vulnerable; or 3) security critical information is in their surrounding context and unseen from the function itself, e.g., calculation of complicated pointer offsets sometime is done via a separate function, where no pointer metrics can be inferred, thus resulting in a lower vulnerability score. For the first case, such vulnerabilities are generally hard to identify via static analysis, and should not be a concern of our approach. Case two is also irrelevant to the validity of our approach. A mitigation for the third case is to include taint information to our vulnerability metrics, as will be discussed in §V.

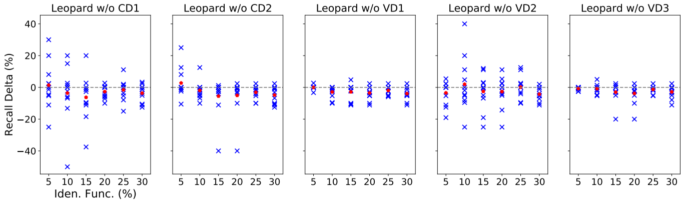


Fig. 5: Sensitivity Analysis Results of Metrics

False Positive Analysis. Balancing the generality, accuracy and scalability is always a very challenging task for static analysis. Since LEOPARD is designed to reveal general vulnerabilities, it is impossible to avoid false positives. However, LEOPARD aims to assist vulnerability assessment rather than a stand alone static analysis tool. False analysis is therefore not a critical criteria for evaluating its capability. Furthermore, some vulnerabilities are previously patched in history, secretly patched [70] or currently unexposed, and it is impossible to confirm whether they are indeed false positives. This is also reflected in the experiments in § IV-F, where new vulnerabilities have been found in the reported potential vulnerable functions.

Our binning-and-ranking approach is effective, i.e., identifying 20% of functions as vulnerable to cover 74.0% of vulnerable functions on average. Such a small portion of functions can be very useful for security experts, as will be shown in our application of LEOPARD in § IV-F. Besides, LEOPARD outperforms machine learning-based techniques and static analysis-based approaches.

D. *Sensitivity of the Metrics (Q3)*

To evaluate the sensitivity of the complexity and vulnerability metrics to our framework, we removed one of the dimensions of the complexity and vulnerability metrics from LEOPARD, and then ran LEOPARD on all the projects. We show the sensitivity results of complexity metrics and vulnerability metrics in Fig. 5. The x-axis and y-axis represent *Iden. Func.* and the *delta* of recall (i.e., *Cov. Vul. Func.*) compared to LEOPARD with all metrics. After removing one dimension of metrics, the recall delta of each project when identifying certain percentage of functions are labeled by blue cross marks, where positive delta means improvement in performance, and negative ones means degradation. The red dots are average recall delta among all 11 projects.

We can see from Fig. 5 that, basically, there are much more degradation than improvement when removing any dimension of metrics. Moreover, the average recall deltas across projects are negative for *Iden. Func.* at 15%, 20%, 25%, and 30%

in all five experiments, i.e., less vulnerable functions are covered when the same percentage of functions is identified as vulnerable. Some improvement of average recall delta at 5% and 10% actually results from some relatively large improvements of only a few projects. Specifically, most significant degradation occurs when the cyclomatic complexity metrics (i.e., CD1) is removed, and most significant average degradation occurs when the loop structure metrics are removed, which indicates they make substantial contribution to our framework. It also proves the necessity of our binning strategy. With the above observation, we can conclude that all dimensions of our complexity and vulnerability metrics contribute to the effectiveness of LEOPARD, but complexity metrics contribute the most; and it is difficult or even impossible to derive an optimal model for the metric combination that can work well for all ranges of *Iden. Func.* for all projects. Hence, we design a generic but not optimal model that treats each metric equally.

> Complexity metrics significantly contribute to LEOPARD; and it is difficult to derive an optimal metric model that works for all projects, which motivates our generic model without sacrificing much effectiveness.

E. Scalability of Our Framework (Q4)

To evaluate the scalability of our framework, we collected the time of extracting complexity and vulnerability metrics and the time of identifying potentially vulnerable functions by LEOPARD. The detailed results are reported at our website [6]. The time used to build the code property graph and query the graph to obtain metric values depends on the number of functions in each project. For small-scale projects, it respectively takes 2 and 45 minutes to build and query the graph; and it takes hours for large-scale projects (i.e., `Wireshark` and `Linux`). It takes less than 50 seconds to identify 100% functions even for `Linux`. These results demonstrate that our framework scales well for large-size projects like `Linux`. For machine learning-based techniques, GB on average takes 9 minutes to train the model and make the prediction for each project, and RF takes 5 minutes. Considering they also depend on the metrics calculation, LEOPARD is more efficient. S*** basically takes several minutes to finish the static analysis but requires the project to be well compiled and built, and fail to handle `Linux`. The lightweight static scanner Cppcheck shows comparable performance as LEOPARD.

> Our framework scales well and can be applied to large-scale applications like `Linux`.

F. Application of LEOPARD *(Q5)*

Manual Auditing. Code review is a popular approach for vulnerability hunting. In this section, we demonstrate the role that LEOPARD plays in helping security experts to hunt vulnerabilities with a case study of FFmpeg 3.1.3. In order not to overwhelm the security expert, we showed the top 1% candidates with LEOPARD, which is a list of 128 functions with detailed complexity and vulnerability metric scores, as well as the specific variables involved in the metrics, e.g., the variables involved in control predicates. The security expert is experienced with code review and is familiar with the basic implementation and code structures of *FFmpeg*. He firstly grouped the functions into different modules and chose *libavformat* as the target, which is the module responsible for the streaming protocols and conversion, and has been prone to vulnerabilities in history. Among all 128 functions, 13 of them are in *libavformat*. He spent one day to find a *divide-by-zero* bug in one of the functions, with CVE-2018-14394 assigned. Intuitively, he thinks the maximum of data-dependent control structures metrics (with the variables involved) more interesting, as he can be guided to trace backward and/or forward the data flow of these sensitive variables. Detailed discussion about the aforementioned case can be found at our website [6].

TABLE VII: Zero-Day Vulnerabilities in PHP

Module	CVE-ID	Type	Reproducible?	
			32-bit	64-bit
php::mbstring	CVE-2017-9224	stack out-of-bound read	✓	✓
php::mbstring	CVE-2017-9225	heap out-of-bound write	✓	✗
php::mbstring	CVE-2017-9226	heap out-of-bound write	✓	✓
php::Zend	CVE-2017-9227	stack out-of-bound read	✓	✓
php::mbstring	CVE-2017-9228	heap out-of-bound write	✓	✓
php::mbstring	CVE-2017-9229	invalid dereference DoS	✓	✓

Directed Fuzzing. As discussed in § III, LEOPARD can supply targets for directed fuzzing. Experimentally, we ran LEOPARD on `PHP 5.6.30` (a popular general-purpose scripting language that is especially suited to web development) and identified around 500 functions as potentially vulnerable. Notice that `PHP` is used by more than 80% of all the websites, and `5.6.30` is the current stable version. Thus `PHP` is well-tested by its users, developers, and security researchers, and it is difficult to find vulnerabilities. We selected top 500 functions reported by LEOPARD as the target sites for Hawkeye for bug hunting. We divided `PHP` into several modules based on its architecture and focused on the functions in the modules (e.g., mbstring and Zend) that are related to file system and network data as they are often reachable through entry points. We excluded the functions in those well-fuzzed modules (e.g., SQLite, phar and gd). This manual filtering process is different from manual auditing as the security expert does not pinpoint the vulnerability directly. After 6-hour fuzzing, we discovered six vulnerabilities in `PHP 5.6.30` with details shown in Table VII.

Seed Prioritization. In § III, we also discussed the application of applying the results of LEOPARD to the seed evaluation process during fuzzing. We used LEOPARD to generate function level scores for three real-world open-source projects and utilized the scores to provide guidance to FOT [19]. The three projects are *mjs* [1] (a Javascript engine for embedded systems), *xed* [2] (the disassembler used in *Intel-Pin*) and *radare2* [3] (a popular open source reverse engineering framework). For the experiment purpose, we ran FOT with and without the guidance from LEOPARD for 24 hours and collected the detected crashes.

Table VIII shows the detailed performance differences of FOT with and without LEOPARD. From the results, LEOPARD can help FOT to detect 127% more crashes in 24 hours on average. Finally, seven new bugs are found in *mjs*, seven new bugs are found in *xed*, and a new vulnerability (CVE-2018-

TABLE VIII: Crashes Detected in 24 Hours by FOT with and without the Results from Leopard

Project	mjs	xed	radare2	Average
w/o Leopard	181	720	7	303
with Leopard	**251**	**1800**	**9**	**687**

14017) is exposed in *radare2*.

> These results showed that Leopard can substantially enhance the vulnerability finding for a limited time budget, which is the original purpose of designing Leopard.

V. Metrics Extension

The set of complexity and vulnerability metrics can be refined and extended, to highlight interesting functions via capturing different perspectives. To this end, we have identified the following information to be vital to further improve our findings.

Taint Information. Leveraging taint information will help an analyst to identify the functions that process the external (i.e., taint) input. In general, functions that process or propagate the taint information can be considered quite interesting for further assessment. Hence, incorporating the taint information into vulnerability metrics will further enhance the Leopard's ranking step by assigning more weight (or importance) to the functions that process or propagate the taint information.

Vulnerability History. In general, when a vulnerability is reported, the functions related to the vulnerability will go through an intensive security assessment during the patching process. Hence, such information can be used to refine the ranking by either: (1) giving more importance to recently patched functions due to the verified reachability, with considerable risks of incomplete patch or introducing new issues, or (2) giving low priority to the functions that are patched long before the release of the current version, assuming that the functions have gone through a thorough security assessment and it is not worth the effort to re-assess it.

Domain Knowledge. Domain knowledge can play a vital role in prioritizing the interesting functions for further assessment. Information such as the modules that are currently fuzzed by others or the knowledge about the modules that are shared by two or more projects can be used to refine Leopard's ranking.

VI. Related Work

Here we discuss the most closely related work that aim at assisting security experts during vulnerability assessment.

Pattern-Based Approaches. Pattern-based approaches use patterns of known vulnerabilities to identify potentially vulnerable code. Initially, code scanners (e.g., Flawfinder [5], PScan [8], RATS [9] and ITS4 [64]) were proposed to match vulnerability patterns. These scanners are efficient and practical, but fail to identify complex vulnerabilities as the patterns are often coarse-grained and straightforward. Differently, our approach does not require any patterns or prior knowledge of vulnerabilities.

Since then, security researchers have started to leverage more advanced static analysis techniques for pattern-based vulnerability identification (e.g., [18, 29, 34, 37, 59, 63, 71, 72, 74]). These approaches require the existence of known vulnerabilities or security knowledge as the guideline to formulate patterns. As a result, they can only identify similar but not new vulnerable code. Differently, we do not require any pattern inputs or prior knowledge of vulnerabilities, and can find new types of vulnerabilities.

Besides, several attempts have been made to automatically infer vulnerability patterns (e.g., [41, 62, 73]). While promising, these approaches only support specific types of vulnerabilities, e.g., missing-checking vulnerabilities for [62] and taint-style vulnerabilities for [41, 73]. However, our approach can find new types of vulnerabilities.

Metric-Based Approaches. Inspired by bug prediction [16, 28, 30, 38, 49], a number of advances have been made in applying machine learning to predict vulnerable code mostly at the granularity level of a source file. In particular, researchers started by leveraging complexity metrics [21, 44, 45, 55, 56] to predict vulnerable files. Then, they attempted to combine complexity metrics with more metrics such as code churn metrics and token frequency metrics [26, 31, 43, 47, 48, 52, 54, 54, 57, 58, 58, 65, 79, 81]. Then, advances have been made to use unsupervised machine learning to predict bugs [25, 32, 36, 46, 75, 76, 77, 78, 80] using the similar set of complexity metrics. These approaches use the similar metrics as those in bug prediction, but do not capture the difference between vulnerable code and buggy code, which hinders the effectiveness. Moreover, the imbalance between vulnerable and non-vulnerable code is severe, which hinders the applicability of machine learning to vulnerable code identification. Instead, our approach specifically derives a set of vulnerability metrics to help identify vulnerable functions.

Vulnerability-Specific Static Analysis. Researchers have attempted to detect specific types of vulnerabilities via static analysis; e.g., buffer overflows [24, 82], format string vulnerabilities [24, 53], SQL injections [23, 33, 69], cross-site scripting [23, 33, 35] and client-side validation vulnerabilities [51]. While they are effective at detecting specific types of vulnerabilities, they often fail to be applicable to a wider range of vulnerability types. Moreover, they often require heavyweight program analysis techniques. Differently, our approach is designed to be generic and lightweight.

VII. Conclusions

We have proposed and implemented a generic, lightweight and extensible framework, named Leopard, to identify potential vulnerable code at the function level through two sets of systematically derived program metrics. Experimental results on 11 real-world projects have demonstrated the effectiveness, scalability and applications of Leopard.

VIII. Acknowledgment

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30, Award No. NRF2016NCR-NCR002-026) and administered by the National Cybersecurity R&D Directorate.

References

[1] "mjs," https://github.com/cesanta/mjs, accessed: 2018-04-01.

[2] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, accessed: 2018-04-01.

[3] "radare2: reverse engineering framework," https://github.com/radare/radare2, accessed: 2018-04-01.

[4] (2017) American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[5] "Flawfinder," https://www.dwheeler.com/flawfinder/, 2017.

[6] "Leopard," https://sites.google.com/site/leopardsite2017/, 2017.

[7] "National vulnerability database," https://nvd.nist.gov/, 2017.

[8] "Pscan," http://deployingradius.com/pscan/, 2017.

[9] "Rats," https://code.google.com/archive/p/rough-auditing-tool-for-security/, 2017.

[10] "Cppcheck," http://cppcheck.sourceforge.net/, 2018.

[11] "Cvedetails," http://www.cvedetails.com/, 2018.

[12] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA*, 2011, pp. 12–22.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2329–2344. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134020

[14] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016, pp. 1032–1043.

[15] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities?: A study of the chromium project," in *MSR*, 2015, pp. 269–279.

[16] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7346–7354, 2009.

[17] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *SP*, 2015, pp. 725–741.

[18] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *FSE*, 2016, pp. 678–689.

[19] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: A versatile, configurable, extensible fuzzing framework," in *ESEC/FSE*, 2018, pp. 867–870.

[20] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *CCS*, 2018, pp. 2095–2108.

[21] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, 2011.

[22] G. Coldwind, "How to find vulnerabilities?" http://gynvael.coldwind.pl/?lang=en&id=659, 2017.

[23] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis," in *NDSS*, 2014.

[24] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, 2002.

[25] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *FSE*, 2017, pp. 72–83.

[26] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *QoP*, 2008, pp. 31–38.

[27] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[28] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[29] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," in *ICSE*, 2006, pp. 232–241.

[30] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012.

[31] A. Hovsepyan, R. Scandariato, and W. Joosen, "Is newer always better?: The case of vulnerability prediction models," in *ESEM*, 2016, pp. 26:1–26:6.

[32] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *ICSME*, 2017, pp. 159–170.

[33] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *SP*, 2006, pp. 258–263.

[34] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, "Righting software," *IEEE Softw.*, vol. 21, no. 3, pp. 92–100, 2004.

[35] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *CCS*, 2013, pp. 1193–1204.

[36] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *ESEM*, 2017, pp. 11–19.

[37] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *USENIX Security*, 2005.

[38] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, no. C, pp. 504–518, 2015.

[39] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[40] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[41] I. Medeiros, N. Neves, and M. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities," in *ISSTA*, 2016, pp. 1–11.

[42] G. Meng, Y. Liu, J. Zhang, A. Pokluda, and R. Boutaba, "Collaborative security: A survey and taxonomy," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1:1–1:42, 2015.

[43] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *HotSoS*, 2015, pp. 4:1–4:9.

[44] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *SAC*, 2016, pp. 1415–1421.

[45] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud and Security*, vol. 2013, no. 5, pp. 8–17, 2013.

[46] J. Nam and S. Kim, "Clami: Defect prediction on unlabeled datasets (t)," in *ASE*, 2015, pp. 452–463.

[47] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *CCS*, 2007, pp. 529–540.

[48] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *MetriSec*, 2010, pp. 3:1–3:8.

[49] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, 2017.

[51] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *NDSS*, 2010.

[52] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[53] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *USENIX Security*, 2001.

[54] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, 2011.

[55] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *ESEM*, 2008, pp. 315–317.

[56] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *QoP*, 2008, pp. 47–50.

[57] Y. Shin and L. Williams, "An initial study on the use of execution complexity metrics as indicators of software vulnerabilities," in *SESS*, 2011, pp. 1–7.

[58] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?'," *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[59] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: Finding missing security checks when you do not know what checks are," in *OOPSLA*, 2011, pp. 1069–1084.

[60] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[61] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *SP*, 2013, pp. 48–62.

[62] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "Autoises: Auto-

matically inferring security specifications and detecting violations," in *USENIX Security*, 2008, pp. 379–394.

[63] J. Vanegue and S. K. Lahiri, "Towards practical reactive security audit using extended static checkers," in *SP*, 2013, pp. 33–47.

[64] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *ACSAC*, 2000, pp. 257–267.

[65] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *ISSRE*, 2014, pp. 23–33.

[66] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *SP*, 2010, pp. 497–512.

[67] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS*, 2013, pp. 511–522.

[68] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," in *FSE*, 2016, pp. 61–72.

[69] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX Security*, 2006.

[70] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries - towards understanding the pain and pills," in *ICSE*, 2017.

[71] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *SP*, 2014, pp. 590–604.

[72] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *ACSAC*, 2012, pp. 359–368.

[73] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *SP*, 2015, pp. 797–812.

[74] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *CCS*, 2013, pp. 499–510.

[75] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *ESEM*, 2017, pp. 344–353.

[76] M. Yan, X. Zhang, C. Liu, L. Xu, M. Yang, and D. Yang, "Automated change-prone class prediction on unlabeled dataset using unsupervised method," *Information and Software Technology*, vol. 92, pp. 1–16, 2017.

[77] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *FSE*, 2016, pp. 157–168.

[78] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *ICSE*, 2016, pp. 309–320.

[79] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining software metrics and text features for vulnerable file prediction," in *ICECCS*, 2015, pp. 40–49.

[80] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? an examination of cross-project defect prediction," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, p. 1, 2018.

[81] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *ICST*, 2010, pp. 421–428.

[82] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *FSE*, 2004, pp. 97–106.