A Clairvoyant Approach to Evaluating Software (In)Security

Bhushan Jain  
bhushan@cs.unc.edu  
The University of North Carolina at Chapel Hill

Chia-Che Tsai  
chitsai@cs.stonybrook.edu  
Stony Brook University

Donald E. Porter  
porter@cs.unc.edu  
The University of North Carolina at Chapel Hill

ABSTRACT

Nearly all modern software has security flaws—either known or unknown by the users. However, metrics for evaluating software security (or lack thereof) are noisy at best. Common evaluation methods include counting the past vulnerabilities of the program, or comparing the size of the Trusted Computing Base (TCB), measured in lines of code (LoC) or binary size. Other than deleting large swaths of code from project, it is difficult to assess whether a code change decreased the likelihood of a future security vulnerability. Developers need a practical, constructive way of evaluating security.

This position paper argues that we actually have all the tools needed to design a better, empirical method of security evaluation. We discuss related work that estimates the severity and vulnerability of certain attack vectors based on code properties that can be determined via static analysis. This paper proposes a grand, unified model that can predict the risk and severity of vulnerabilities in a program. Our prediction model uses machine learning to correlate these code features of open-source applications with the history of vulnerabilities reported in the CVE (Common Vulnerabilities and Exposures) database. Based on this model, one can incorporate an analysis into the standard development cycle that predicts whether the code is becoming more or less prone to vulnerabilities.

ACM Reference format:
https://doi.org/10.475/123_4

“If you can not measure it, you can not improve it.”  
—Lord Kelvin (Sir William Thomson)

1 INTRODUCTION

Evaluating software security is hard. When a developer makes a change to her codebase, she needs a way to validate that the change improves overall security, or at least is unlikely to have opened up new vulnerabilities. Although formal methods to prove systems correct are making great strides [39, 44, 63], it will be some time, if ever, before most software is verified.

In the meantime, developers are stuck with several unappealing options. First, developers can hope for the best, and fix vulnerabilities reactively (i.e., after systems have been compromised). Second, developers can use a number of bug finding or code quality assessment tools [9, 11, 16, 17, 25, 26, 28, 31, 34, 40, 50, 54, 67]; clearing all warnings from these tools will certainly improve code quality, but not all code quality issues pose equal security risks. Finally, developers can use noisy metrics, like total lines of code, which have certain “local maxima” (e.g., delete everything), and do not improve both functionality and security.

In the research community, system researchers primarily use noisy metrics, such as lines of code, to evaluate relative security of software products or prototypes. We surveyed the papers published in several top systems conferences, including CCS, PLDI, SOSP, ASPLOS, and Eurosys, and found 384 papers using lines of code to evaluate the security of their solutions, or to indicate the insecurity of alternatives, as shown in Figure 1. 116 papers in our survey use reports from the CVE (Common Vulnerabilities and Exposures) databases [5], to analyze software vulnerability. Only 31 papers have formally verified or mathematically proved the security of their solutions. Thus, the de facto metric for security in systems research is counting lines of code.

This paper argues that the computer science community now has all of the right tools needed to design a better, empirical security evaluation. Even if one can’t prove the absence of vulnerabilities, it is possible to estimate the level of security (or insecurity) in a program, by approximating the likelihood of having vulnerabilities. A large body of security research has shown various correlations between the occurrence of certain types of vulnerabilities and code properties such as the choice of programming languages, code complexity, and length of the program. [18, 21, 25, 28, 31, 33, 35]. Collectively, these code properties represent a fairly wide range of noisy, but meaningful signals; we argue that distilling a more
precise prediction from this data is precisely what machine learning is good at.

In order to make this approximation precise, we propose to use machine learning techniques to identify correlations between the code properties and the vulnerabilities reported in the CVE database. In our prediction model, the severity and classification of vulnerabilities from the CVE database is a vast ground truth data, which can help the machine learning engine correlating the occurrence of vulnerabilities and various program properties. Using this model, we select open-source applications from the CVE database that have a converging history of vulnerability reporting. Based on the CVE tuples of each application, we can learn the correlation between code properties and the existence of vulnerabilities, with cross validation within the ground truth.

For such a metric to succeed, it must be reasonably predictive, and encourage improvement over previous versions of the code, or in selecting among competing options. For instance, in selecting between two library implementations for use in a web service, our proposed metric would identify which is less likely to have vulnerabilities. Since the prediction model is trained offline, and the analysis of program properties can be mostly automated, the security evaluation requires very little effort from the developers.

2 BETWEEN A ROCK AND A HARD PLACE

Although we are perhaps within striking distance of the strong assurances of formal methods, it will be some time before most code is proved correct. In the mean time, the state of the art amounts to testing. This section motivates the need for a metric that can capture the level of risk in imperfect software.

2.1 Formal Verification

One approach to ensure correct behavior of a program is formal verification. Formal verification generally proves that an implementation satisfies a model, or specification, of the program’s expected behavior. Assuming the specification is written to preclude insecure behavior (for some definition of secure), a formally-verified codebase will also be secure. There are two main approaches: developers either write a mathematical proof of the program specifications (deductive model) [20, 36, 42, 43, 51], or to specify a finite state machine model of program execution for exhaustive checking (model checking) [19, 24, 32, 35, 50].

Formal verification is a “holy grail” for developers; as E.W. Dijkstra pithily observed, “program testing can be used very effectively to show the presence of bugs but never to show their absence” [7]. Using formal verification, a developer can actually prove the absence of bugs in her program. However, there are two practical obstacles for verification. The simple problem is that the tools are relatively nascent, and only a few researchers have the expertise to use them [30]. Moreover, several systems papers on verification propose enhancements to the tools or proof techniques themselves to make the proof go through [38, 44, 57, 62]. A deeper challenge is that designing a comprehensive specification that precludes all attack vectors is challenging and requires a different set of skills than writing an implementation. Writing a specification is essential however, as it is questionable the degree to which one can reason about system security without some implicit notion of correct behavior; crisply articulating what is “correct” is generally reported to be a useful exercise.

2.2 Testing

Penetration testing is an empirical alternative to formal verification. The developers can recruit a team of expert attackers that test likely attack vectors. Like any other testing strategies, bugs may be overlooked, and the quality of the test depends heavily on the skill of the testers. One of the successful example of this type of security evaluation is the annual hacking contest Pwn2Own [14]. However, the cost of “bug bounties” can be high. For example, Google offered $1 million to hackers who can produce zero-day exploits against its Chrome browser [13]. Furthermore, auditors can do a code review to check for any vulnerabilities, but is a labor-intensive process. Auditing ensures that proper security standards are met, which should reduce the existence of vulnerabilities in the code. The goal of the penetration testing is to find vulnerabilities, while the goal of the auditor is to ensure safe practices are followed to prevent vulnerabilities.

In order to evaluate the security improvement in the newer versions of the program, the developers often test that the vulnerabilities reported on the older versions of a program are fixed in the newer version. This is a natural extension to regression testing in a continuous integration (CI) workflow, and easy for developers to adopt. Like other testing approaches, this approach is only as comprehensive as the unit tests.

3 NO SINGLE METRIC IS COMPREHENSIVE

A number of approaches use code properties as a way to estimate the likelihood of having vulnerabilities in a program. The code properties that are commonly used by software or research projects include LoC, binary size, and number of interfaces. Researchers have also developed software engineering techniques to approximate the complexity of a program, such as using McCabe’s cyclomatic complexity metric [47] or Halstead complexity measures [37]. In addition, there is a long line of research using code properties to indicate “code smell” [45, 46, 49, 55, 58, 64, 65, 68]—symptoms or patterns of bad coding practice, such as lines of comments or numbers of long methods. The underlying assumption is that the number of vulnerabilities is generally correlated with these code properties.

3.1 Lines of Code is a Bad Metric

The conventional wisdom is that the number of bugs is correlated with LoC or binary size. As a result, LoC or binary size is frequently used in system research to evaluate software prototypes. And there is both intuitive and anecdotal appeal for this notion. However, not all bugs are vulnerabilities. A prior study [23] on the Chromium project [2] shows that bug estimates are not direct estimates of vulnerabilities. A bug may not change the state of the program in a way that violates security-related invariants, or may not be triggered by interfaces exposed to external attackers.

To test whether this conventional wisdom applies to vulnerabilities, we surveyed 164 open source applications from the CVE database, all of which have at least 5 years of CVE history (i.e., the
time of the newest CVE report minus the time of the oldest CVE report). We calculated the lines of code using cloc [29]. We categorized the applications by the languages in which they are primarily programmed: 126 are primarily in C, 20 are primarily in C++, 6 are primarily in Python, and 12 are primarily in Java. Figure 2 shows the correlation between the number of vulnerabilities (regardless of the severity) and the total LoC in applications.

We observe that programs with large numbers of vulnerabilities tend to have large code bases. However, there is only a weak correlation between LoC and the number of vulnerabilities. Only when one buckets application sizes and vulnerability counts by orders of magnitude is there a weak correlation. Specifically, the coefficient of determination (i.e., $R^2$) is 24.66%—meaning 75.34% of the data set cannot be explained by the trend line, even when bucketed by order of magnitude. Thus, this data does not support the notion that simply comparing lines of code within the same order of magnitude is a meaningful indicator of security.

People often argue that some languages are inherently more secure. Although some common bug patterns, such as pointer errors, are precluded by higher-level languages, this evidence does not support that Python is any less prone to errors than C or C++. In this data set, the sample size for Java is admittedly small (only 12 projects), but the Java projects do have a lower number of vulnerabilities. For other languages, no significant trend can be observed from the study.

The study shows a lesson: using LoC for security evaluation is not statistically significant if the difference is within one or two orders of magnitude. LoC is too noisy to make any high-quality predictions, and can lead to naive or facile conclusions. The security of a program is under the influence of a number of factors, such as expertise of the programmers, code maturity, and level of code review.

### 3.2 Other Metrics Are Also Noisy

Similar to lines of code, other code properties, such as McCabe’s cyclomatic complexity metric [47], are also noisy indicators of security problems. The cyclomatic complexity is measured as the number of linearly independent paths through a program’s source code. This complexity increases with the number of conditional statements, and their nesting. Similar to LoC, cyclomatic complexity is also weakly correlated to the number of vulnerabilities reported in CVE database, as we demonstrate in Figure 3.

Apart from measuring code properties, other models have been proposed to measure software security. Some of the standards regarding security design metrics are NIST 800-55 [52], Common Criteria [54], and ISO/IEC 27004 [8]. However, as Wang explains, “these specifications are either too broad without precise definitions or too specific to cover all the security situations” [66]. Most of the standards are qualitative, subjective, and defined without a formal model, specifying different levels of trust, and different interpretation of these levels. Alshammari et al. [16] designed security metrics for object-oriented programs to measure accessibility of objects, using information flow tracking to observe interactions among classes. However, this metric is limited to object-oriented designs that only limits the interactions among different objects and classes.

In addition to design standards, some individual aspects of a program are also used to measure software security. Howard et al. [41] calculate the attackability of a system as Relative Attack Surface Quotient (RASQ) by measuring attack surface in terms of the resources available to the attacker, the communication channels, and access rights. As the authors state, these measure some dimensions of attack surface, and are not necessarily all, or even the right, dimensions. Their score is only relative to another system, and limited by the specific system configuration. Nicol et al. [50] evaluate system security from model-based techniques for evaluating system dependability. While dependability can be one of the factors affecting security of a software, it is not the only factor governing security of a software. Wang et al. [67] combine the Common Vulnerability Scoring System (CVSS) score of all the known CVE reports of a software, to assign a final security metric score. While, this metric can be a good way to compare the security of two pieces of software, this technique does not consider unknown vulnerabilities, nor is their score dependent on any other information than the CVSS score.
We can draw additional code metrics from the security literature. We can estimate how difficult it is to attack a program by building with vulnerabilities, albeit by a different formula. Our position is vulnerability that subverts the security of the program. Therefore, vulnerabilities.

that can be triggered by specific ranges of inputs.

calculate the number of different execution paths in a program using symbolic execution [22] or abstract interpretation [27], we numbers of calling and returning targets in a program. Moreover, other parts of the code. Control flow analysis [15] can determine numbers of expressions or functions influencing the execution of an attack-graph [60].

all the attackers need to penetrate a program is one high-impact on a program, or defend against the attack. However, in reality, evaluate how difficult it is to either launch a specific type of attack use Relative Attack Surface Quotient (RASQ) [41]. We can also use Relative Complexity (RC) as a type of bugs, some of these code properties may also correlate with vulnerabilities, albeit by a different formula. Our position is that a weighted aggregation of multiple metrics can provide a more precise estimation of potential vulnerabilities.

One prior project [61] has shown the predictability of multiple code properties for software vulnerability, in a limited scope. Shin et al. evaluate complexity, code churn, and developer activity metrics as indicators of software vulnerabilities in Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. They are able to predict 80% of the vulnerable files, by taking into account most basic properties of code files such as LoC, number of functions, number of declarations, lines of preprocessed code, number of branches, and number of input and output arguments to a function. Even though this study only predicts vulnerable files to be inspected, it shows that these attributes are correlated with the existence of vulnerabilities.

We propose expanding the study of Shin et al. to a wider range of software and code properties. Learning a prediction function from a larger feature vector that indicates the potential for vulnerabilities can only improve the prediction accuracy. Our goal is to yield a useful security metric that can predict the number, classification, and severity of vulnerabilities, rather than just files to be inspected.

4 MAYBE MORE METRICS?

Metrics like lines of code, cyclomatic complexity, and other code properties are noisy indicators and only measure specific aspects of a program. Most of these properties have indicated the likelihood of certain types of bugs. However, because we treat vulnerabilities as a type of bugs, some of these code properties may also correlate with vulnerabilities, albeit by a different formula. Our position is that a weighted aggregation of multiple metrics can provide a more precise estimation of potential vulnerabilities.

5 STRENGTHS AND WEAKNESSES

One prior project [61] has shown the predictability of multiple code properties for software vulnerability, in a limited scope. Shin et al. evaluate complexity, code churn, and developer activity metrics as indicators of software vulnerabilities in Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. They are able to predict 80% of the vulnerable files, by taking into account most basic properties of code files such as LoC, number of functions, number of declarations, lines of preprocessed code, number of branches, and number of input and output arguments to a function. Even though this study only predicts vulnerable files to be inspected, it shows that these attributes are correlated with the existence of vulnerabilities.

We propose expanding the study of Shin et al. to a wider range of software and code properties. Learning a prediction function from a larger feature vector that indicates the potential for vulnerabilities can only improve the prediction accuracy. Our goal is to yield a useful security metric that can predict the number, classification, and severity of vulnerabilities, rather than just files to be inspected.

4.1 Finding More Code Properties

We can draw additional code metrics from the security literature. For example, to measure the attack surface of a program, one can use Relative Attack Surface Quotient (RASQ) [41]. We can also evaluate how difficult it is to either launch a specific type of attack on a program, or defend against the attack. However, in reality, all the attackers need to penetrate a program is one high-impact vulnerability that subverts the security of the program. Therefore, we can estimate how difficult it is to attack a program by building an attack-graph [60].

We can collect additional metrics from static analysis that are likely relevant. For instance, data flow analysis [56] can determine numbers of expressions or functions influencing the execution of other parts of the code. Control flow analysis [15] can determine numbers of calling and returning targets in a program. Moreover, using symbolic execution [22] or abstract interpretation [27], we can calculate the number of different execution paths in a program that can be triggered by specific ranges of inputs.

4.2 Leveraging Bug-finding Tools

We can also extract information from existing bug-finding tools. In general, the concern with many bug-finding tools is a high false positive rate; it is unclear whether a machine learning tool will be sufficient to separate the wheat from the chaff without human intervention. A simple way is to feed the bug reports or count of bug types into the machine learning engine. Although finding bugs is a different goal from security evaluation, using the results of bug-finding tools as code properties can amortize the inaccuracy of locating bugs, and potentially lead to the discovery of hidden correlation with other types of vulnerabilities.

A number of research projects have developed bug or defect finding techniques, which analyze the source code or the binary to find known code properties patterns or apply a model. Some of these tools are language-specific. Java specific tools such as PMD [11], FindBugs [40], and JLint [9] use syntactic bug pattern detection, while ESC/Java [34] uses theorem proving, and Bandera [26] uses model checking. Lint [17] is a C program checker that finds program constructions that can lead to future errors. Rutar et al. [59] compare bug finding tools for JAVA, and design a meta-tool that combines the output of all other tools.

Some researchers use program specifications and models to find bugs. Dolby et al. [31], choose to encode a program as a relational logic, and then use a constraint solver to find specification violations. Couto et al. [28], study the causal relationships between quality metrics and bugs. Chen et al. [25] identify safe programming practices, encode them as safety properties, and verify whether these properties are obeyed by the programs. We can use these bug-finding tools as independent inputs to the machine learning block.

5 EXTRACTING MORE SIGNAL FROM MORE NOISE

In this section we propose a system design to create and apply our proposed security metric in future work. Given a codebase, the metric will predict the occurrence of vulnerabilities by severity and classification. The steps for building the system include constructing a testbed for data collection, using machine learning to train the prediction model, and then letting developers apply the metric to evaluate their code.

5.1 The Testbed

To train a stable model for prediction, we need a representative dataset of software vulnerabilities. We propose to collect the past vulnerabilities from the CVE (Common Vulnerabilities and Exposures) database [5]. The CVE database reports more than 80,000 vulnerabilities in ~400 applications and systems. CVE exports a data set that is ready for analysis; for each vulnerability, its classification, impact, and severity is represented by a metric called Common Vulnerability Scoring System (CVSS) (the current version is v3.0) [3]. The CVSS scores are based on a lot of different factors including attack vector (AV), attack complexity (AC), privileges required (PR), confidentiality impact (C), integrity impact (I), and exploit code maturity (E).

In the CVE database, some applications have been maintained and debugged for decades, whereas other applications are relatively
A Clairvoyant Approach to Evaluating Software (In)Security

HotOS’17, May 2017, British Columbia, Canada

5.3 Using the Metric

The outcome of the training phase is a classifier, which predicts the number, severity, classification, and impact of vulnerabilities, for any application. We envision software developers applying this classifier to their source code. The automated testbed can be used for collecting code properties in developer’s codebase. Based on these code properties, the classifier can give the developer an evaluation of, say, whether a code change has raised or lowered the risk than the previous version of the code.

Each weight in the trained model shows the importance of the corresponding code property to the predicted vulnerability. For instance, the classifier can predict the expected risk of an attack via the network. Properties that heavily contribute to a given result can be flagged for developer attention. Developers can use these hints to prioritize the improvement of code properties or choosing a defense mechanism, such as applying bound checking if there is high risk of buffer overflow, or placing the application behind firewall or intrusion protection if a network attack is predicted.

One potential improvement is to collect dynamic traces; dynamic properties of a program may further yield additional insights or accuracy. For ease of deployment and integration with current development tools, we focus on static analysis.

An important question for future work is: can we use the same approach of evaluating application programs to evaluate whole systems? We expect that total system security is dependent upon the weakest link, although factors such as which applications are network-facing have a role as well. Similarly, it is challenging to model areas of containment, such as when a hardware protection boundary (i.e., a “ring” in x86) is crossed, or when services are distributed across multiple, unprivileged users, as in Android. A goal for future work is to apply the metric in to a VM or Docker image, capturing the risk for not just the application, but its supporting infrastructure.

6 CONCLUSION

Developers lack good tools for evaluating whether a code change is likely to improve or reduce the probability of an exploitable vulnerability. Most testing and code quality tools are best-effort.
Formal approaches are beginning to come to maturity, but still require a high degree of sophistication. We present a new approach to security evaluation that is easy for the average developer to employ. This work represents a step towards a more quantitatively-driven approach to measuring risk in software engineering. In the absence of perfect code, code can only become more reliable if we can estimate imperfection.

In future work, we will flesh out the model and analyze against a corpus of programs, as well as seek to validate that the approach improves security. An ongoing challenge for the work is to extend the limited data sets; although CVE data makes this work possible, the data collected is limited to programs that have been used or studied sufficiently well to warrant CVE reporting. Another challenge is translating the metric to actionable code changes; we expect that one might be able to identify individual code metrics that contribute to this risk and work from there. Alternatively, one might use these metrics to focus the effort of bug-finding tools for deeper analysis on particularly risky code, or to focus additional testing effort.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments on the work. This work was supported in part by NSF grants CNS-1405641, CNS-1161541, and VMware.

REFERENCES


