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:
Bhushan Jain, Chia-Che Tsai, and Donald E. Porter. 2017. A Clairvoyant Approach to Evaluating Software (In)Security. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 7 pages.
https://doi.org/10.1145/3102980.3102991

"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.

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3102991

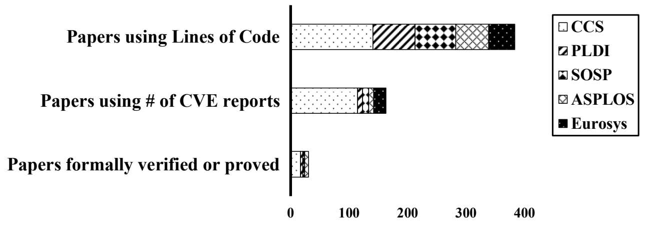

Figure 1: Numbers of papers, in top system proceedings, using lines of code, numbers of past CVE reports, or whether or not the work is formally verified or mathematically proved, to evaluate security or indicate insecurity.

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 programing languages, code complexity, and length of the program. [18, 21, 25, 28, 31, 33, 53]. 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 applications, 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

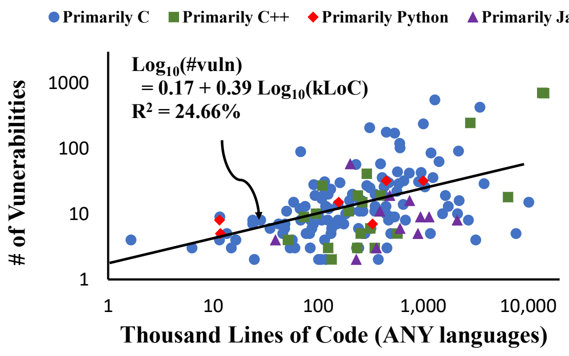

Figure 2: Lines of code plotted against number of vulnerabilities for open-source applications with more than 5-year history in the CVE database. The applications are categorized by the primary programming languages used (either C, C++, Java, or Python).

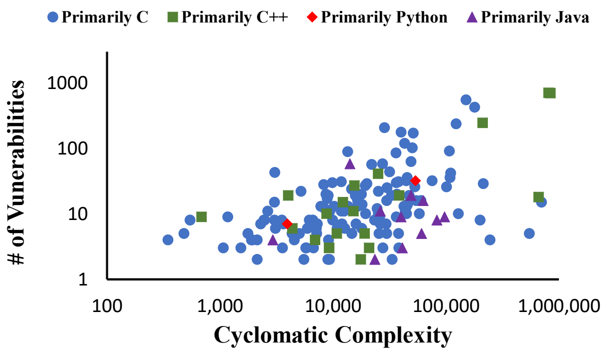

Figure 3: Cyclomatic complexity metric [47] plotted against number of vulnerabilities.

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.

Mellado et al. [48] compare software design security metrics based on different characteristics including authenticity, confidentiality, detection of attacks, and commercial damage. They explain in detail the shortcomings of these security metrics. They observe that the most comprehensive metric that covers all the characteristics is CVSS score. While there exists a few standards and metrics for security evaluation, all of them are either too difficult to use objectively, too specific, or focus on only one specific security relevant attribute of the program.

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.

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

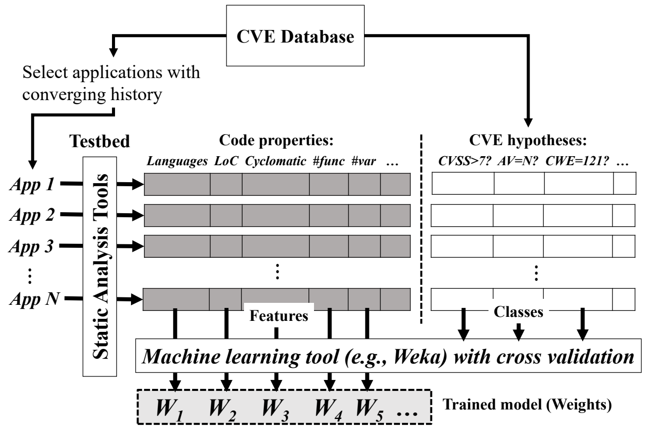
CVE Database
Select applications with converging history
Testbed
Static Analysis Tools
App 1
App 2
App 3
App N
Code properties:
Languages LoC Cyclomatic #func #var ...
CVE hypotheses:
CVSS>7? AV=N? CWE=121? ...
Features
Classes
Machine learning tool (e.g., Weka) with cross validation
W1 W2 W3 W4 W5 ...
Trained model (Weights)

Figure 4: The training phase of our security evaluation model.

immature. Our study will focus on open-source applications which have at least a 5-year history in the CVE database—the same set of applications shown in Figure 2 and 3. Because of the number of features (i.e., code properties) used in the training, we want a sufficiently large quantity of application samples and attributes that the generated model does not overfit. We currently use a training data set of 5,975 vulnerabilities reported for the 164 selected applications, collected as of April 2017. It is unclear how many samples will be enough, although this set will grow over time.

We also need an automated framework to collect all the code properties from the sample applications. Several tools are automatable and are extensible collecting additional code information. For example, CCCC [1] and Metrix++ [10] analyze C/C++ and/or Java code and generate the distribution of both LoC and the cyclomatic complexity metric. Metrix++ is extensible to collect other code properties by specifying regular expressions to search.

5.2 Training the Model

Once we have determined the code properties of our sample applications, we can use these code properties as inputs to train our model of predicting vulnerabilities. The classes to predict include the classifications from the CVE database, such as the weakness types defined by Common Weakness Enumeration (CWE) [4], and CVSS factors, including attack vector (AV), attack complexity (AC), privileges required (PR), impact on confidentiality, integrity or availability (C/I/A), and the aggregated CVSS score.

Our approach is similar to the prior work which uses machine learning to overcome the common weakness of the bug-finding tools—too many false positives. For instance, Brun et al. [21] generate the models of fault-revealing program properties, and apply to user-written code to predict whether the code contains errors. Similarly, Zeng [69] devised a machine learning approach to combine three different bug finding tools in Java.

We use machine learning to train a series of hypotheses on the sample applications: For example, how many high-severity vulnerabilities exist in an application (i.e., *CVSS > 7*)? Does an application contain any vulnerabilities that are accessible from the network (i.e., *Attack Vectors = N*)? Does an application suffer any stack-based buffer overflow (i.e., *CWE = 121* [6])? The data set that we use for training maps the answers of these hypotheses on our sample applications, to a large quantity of code properties collected by our testbed. Using the data set as a *ground truth*, we can train a model that predicts the results of these hypotheses on an unknown applications. Figure 4 shows a high-level overview of the training phase of our security evaluation model.

A data mining tool, such as Weka [12], can then train the *weights* that correlate code properties to these hypotheses. The primary challenge on building this metric will be to refine the trained model, including filtering features that are irrelevant to the prediction, determining necessary data transformation for numeric features, and tuning the parameters to the learning algorithms. Using the trained model, we can present a prediction of unknown vulnerabilities, which can empirically compare two programs, and is generated based on a vector of code properties.

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

[1] CCCC - C and C++ code counter. http://cccc.sourceforge.net/.
[2] Chromium - the chromium projects. https://www.chromium.org/Home.
[3] Common vulnerability scoring system v3.0: Specification document. https://www.first.org/cvss/specification-document.
[4] Common weakness enumeration (CWE). https://cwe.mitre.org/data/index.html.
[5] CVE - common vulnerabilities and exposures. https://cve.mitre.org/.
[6] CWE-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html.
[7] E.W. Dijkstra archive: On the reliability of programs. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html.
[8] ISO/IEC 27004:2016: Monitoring, measurement, analysis and evaluation. http://www.iso27001security.com/html/27004.html.
[9] JLint. Online at http://artho.com/jlint/.
[10] Metrix++ project. metrixplusplus.sourceforge.net/.
[11] PMD. Online at https://pmd.github.io/.
[12] Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/.
[13] Chrome owned by exploits in hacker contests, but google's $1m purse still safe | wired. March 2012.
[14] Pwn2Own 2016: Windows, OS X, Chrome, Edge, Safari all hacked - gHacks tech news. March 2016. (Accessed on 04/25/2017).
[15] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.
[16] B. Alshammari, C. Fidge, and D. Corney. Security metrics for object-oriented class designs. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 11–20. IEEE, 2009.
[17] F. ans Kunst. Lint, a c program checker. 1988.
[18] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.
[19] R. Barbuti, C. Bernardeschi, and N. De Francesco. Checking security of java bytecode by abstract interpretation. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, 2002.
[20] Y. Bertot, G. Huet, P. Castéran, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2013.
[21] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 480–490. IEEE Computer Society, 2004.
[22] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
[23] F. Camilo, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 269–279. IEEE, 2015.
[24] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *NDSS*, volume 4, pages 171–185, 2004.
[25] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS*, pages 235–244. ACM, 2002.
[26] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, et al. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.
[27] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
[28] C. Couto, P. Pires, M. T. Valente, R. Bigonha, A. Hora, and N. Anquetil. Bugmaps-granger: A tool for causality analysis between source code metrics and bugs. In *Brazilian Conference on Software: Theory and Practice (CBSoft'13)*, 2013.
[29] A. Danial. cloc. https://github.com/AlDanial/cloc.
[30] D. Dean, S. Gaurino, L. Eusebi, A. Keplinger, T. Pavlik, R. Watro, A. Cammarata, J. Murray, K. McLaughlin, J. Cheng, et al. Lessons learned in game development for crowdsourced software formal verification. *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, 2015.
[31] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a sat solver. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 195–204. ACM, 2007.
[32] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.
[33] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.
[34] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, May 2002.
[35] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV*, pages 3–18. Springer, 1996.
[36] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
[37] M. H. Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.
[38] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.
[39] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, pages 165–181, 2014.
[40] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.
[41] M. Howard, J. Pincus, and J. M. Wing. Measuring relative attack surfaces. In *Computer Security in the 21st Century*, pages 109–137. Springer, 2005.
[42] P. B. Jackson, B. J. Ellis, and K. Sharp. Using smt solvers to verify high-integrity programs. In *Proceedings of the second workshop on Automated formal methods*, pages 60–68. ACM, 2007.
[43] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-aided reasoning: ACL2 case studies*, volume 4. Springer Science & Business Media, 2013.
[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.
[45] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*, pages 77–80. Society Press, 2005.
[46] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, 2004.
[47] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.
[48] D. Mellado, E. Fernández-Medina, and M. Piattini. A comparison of software design security metrics. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 236–242. ACM, 2010.
[49] N. Moha and Y.-G. Guéhéneuc. Decor: A tool for the detection of design defects. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, 2007.
[50] D. M. Nicol, W. H. Sanders, and K. S. Trivedi. Model-based evaluation: from dependability to security. *IEEE Transactions on dependable and secure computing*, 1(1):48–65, 2004.
[51] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
[52] Nist and E. Aroms. *NIST Special Publication 800-55 Rev1 Security Metrics Guide for Information Technology Systems*. CreateSpace, Paramount, CA, 2012.

[53] K. Pan, S. Kim, and E. J. Whitehead Jr. Bug classification using program slicing metrics. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42. IEEE, 2006.

[54] S. C. P. Profile. Common criteria for information technology security evaluation. 2001.

[55] G. Rasool and Z. Arshad. A review of code smell mining techniques. *J. Softw. Evol. Process*, 27(11):867–895, Nov. 2015.

[56] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[57] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53. ACM, 2015.

[58] N. Roperia. *JSmell: A Bad Smell detection tool for Java systems.* California State University, Long Beach, 2009.

[59] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256. IEEE, 2004.

[60] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.

[61] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[62] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *OSDI*, 2016.

[63] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184. ACM, 2015.

[64] S. Slinger, D. Ing, L. M. F. Moonen, S. Slinger, S. Dr, and I. L. M. F. Moonen. *Title: Code Smell Detection in Eclipse*. 2005.

[65] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, 2008.

[66] A. J. A. Wang. Information security models and metrics. In *Proceedings of the 43rd annual Southeast regional conference-Volume 2*, pages 178–184. ACM, 2005.

[67] J. A. Wang, H. Wang, M. Guo, and M. Xia. Security metrics for software systems. In *Proceedings of the 47th Annual Southeast Regional Conference*, page 47. ACM, 2009.

[68] N. Zazworka and C. Ackermann. Codevizard: A tool to aid the analysis of software evolution. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, 2010.

[69] F. Zeng. A machine learning approach to finding bugs.