CodeTrust

Christian Damsgaard Jensen, Michael Bøndergaard Nielsen

CodeTrust
Trusting Software Systems

Christian Damsgaard Jensen and Michael Bøndergaard Nielsen

Department of Applied Mathematics & Computer Science,
Technical University of Denmark, Copenhagen
cdje@dtu.dk

Abstract. The information society is building on data and the software required to collect and analyse these data, which means that the trustworthiness of these data and software systems is crucially important for the development of society as a whole. Efforts to establish the trustworthiness of software typically include parameters, such as security, reliability, maintainability, correctness and robustness.

In this paper we explore ways to determine the trustworthiness of software, in particular code where some of the constituent components are externally sourced, e.g. through crowd sourcing and open software systems. We examine different quality parameters that we believe define key quality indicators for trustworthy software and define CodeTrust, which is a content based trust metric for software.

We present the design and evaluation of a research prototype that implements the proposed metric, and show the results of preliminary evaluations of CodeTrust using well known open source software projects.

1 Introduction

We are increasingly dependent on software in all the systems and infrastructures that weave the fabric of our everyday life. From the systems that we rely on to do our work, the social media that occupy much of our spare time and the Internet of Things that provides the data and control necessary to support the intelligence and convenience that we expect from our modern way of life. As software becomes ubiquituous and we all need more software systems to function normally, there is an increasing pressure to reduce costs, e.g. through the integration of free (as in gratis) open source software in commodity products. The urgency of this problem is probably best illustrated by the HeartBleed [1] vulnerability discovered in the Open SSL library (e.g. used in the popular Apache web server) or the GNU Bash remote code execution vulnerability Shellshock [6], both discovered in 2014. In both cases, software failures had a crippling effect on popular software systems. The HeartBleed vulnerability in Open SSL was present in several versions of the security protocol library, which meant that many manufacturers, who had relied on Open SSL in their software systems, were unable to determine which of their products were actually vulnerable. Analysis of the Gnu Bash source code, showed that the Shellshock vulnerability in the popular Bash command interpreter had existed since September 1989. These examples show that software vulnerabilities may exist in code for a long time before they are discovered, even in otherwise well respected software like OpenSSL and GNU Bash.

Software systems are often based on components and services that integrate externally developed software libraries and execution engines. Establishing the trustworthiness of the components and services that are integrated into a software system is therefore of vital importance to software developers. This requires a consideration of both directly observable software quality attributes, such as size and complexity, and reputation factors, such as the history of vulnerabilities found in the software components and services and the track record of the software developers. In this paper we propose the CodeTrust software trust metric to evaluate the trustworthiness of a given software system. CodeTrust considers the vulnerability history of the specific software (both number and severity of vulnerabilities), the track record of the software's developers and several directly observable software quality attributes.

We have developed a prototype that implements CodeTrust and used this prototype to evaluate a number of well known Open Source Software (OSS) systems. The evaluation shows a significant variation in the CodeTrust metric for different OSS projects, but that missing data leads to some uncertainty, which suggests that it would be interesting to include a confidence score in the CodeTrust metric.

The rest of this paper is organized as follows. Section 2 presents the Heartbleed vulnerability to illustrate several of the concepts that we consider and provide an overall motivation for our work. Related work on software trust metrics and trust systems used in software development is examined in Section—3. We propose the CodeTrust metric in Section 4, where we also discuss different software quality attributes. We implemented a simple prototype to allow a preliminary evaluation of CodeTrust; this evaluation is presented in Section 5. Finally, we present conclusions and directions for future work are presented in Section 6.

2 Motivation

The HeartBleed vulnerability, mentioned in the Introduction, is one of the best known software vulnerabilities in recent years. HeartBleed was caused by a flawed implementation of the Heartbeat Extension [19] in OpenSSL,[1] which supports keep-alive functionality without performing a renegotiation. The vulnerable code allocates a memory buffer for the message to be returned by the heartbeat based on the length field in the requesting message with no length check for this particular memory allocation. The vulnerable implementation allocates this memory,

[1] OpenSSL is a widely used implementation of the Transport Layer Security (TLS) protocol [9], which allows two computers on the Internet to establish secure communication. It is often used by embedded systems and open source software, such as the Apache and nginx webservers which, at the time, powered around two thirds of the websites on the Internet.

without regard to the actual size of that message's payload, so the returned message returned consists of the payload, possibly followed by whatever else happened to be in the allocated memory buffer on the server. Heartbleed is exploited by sending a malformed heartbeat request with a small payload and large length field to the vulnerable party in order to elicit the victim's response. This allows an attacker to read up to 64 kilobytes of the victim's memory that was likely to have been used previously by OpenSSL, e.g. this memory could include passwords of other users who had recently logged onto the webserver. HeartBleed was first disclosed in April 2014, but the bug was introduced in 2012, when the TLS Heartbeat Extension was first standardised. The flaw was introduced by a single programmer, who implemented the extension, and reviewed by one of OpenSSL's four core developers before being integrated into OpenSSL v. 1.0.1; the bug remained unnoticed in subsequent releases of OpenSSL, until OpenSSL v1.0.1g, which corrected the mistake. An article in the online media The Register indicates around 200,000 Internet sites still vulnerable to HeartBleed in January 2017 [17] and a quick search for the vulnerability in the search-engine Shodan indicates that more than 150,000 Internet sites are still vulnerable a year later.

The HeartBleed vulnerability is not only interesting because it affected an important and popular Internet security component, but mostly because it demonstrates how a small mistake in a software component can have major repercussions for global Internet security, i.e. we place a lot of trust in software, which is delivered for free and without warranty.

3 Related Work

Deciding whether code can be trusted has a long history, e.g. in his *Note on the Confinement Problem* from 1973 [13], Butler Lampson examines the problem of externally sourced software libraries leaking confidential data. Subsequent efforts to address this problem include a number of software evaluation criteria that were especially developed for military use [20, 7], which inspired the development of common criteria for security evaluation of software [12]. Common to these evaluation criteria is that they introduce a well defined hierarchy of increasingly secure systems (encoded in a *Security Level*[2]) and that they do not apply to the software in itself, but to installed systems, e.g. the Common Criteria [12] introduces the notion of a Target of Evaluation(ToE) which defines the set of systems and services considered in the evaluation. Moreover, the official certification process according to these criteria requires significant efforts, resources and time, so they are not widely used outside high security environments, such as defence, aerospace or financial services.

Software quality has been studied from many different aspects over the years, such as reliability and fault tolerance [18] to the notion of dependability [3] and

[2] The Common Critera captures the *Protection Profile (PP)* and the *Security Target (ST)* and measures the *Evaluation Assurance Level (EAL)*, which depends on the depth and rigour of the security evaluation.

the necessary consideration of the full tool-chain [23]. In the following, we limit our presentation to a few efforts that have influenced our work the most.

The Trusted Software Methodology (TSM)[2] found 44 Trust Principles that determine the quality of software. These principles combine aspects of both security and software engineering as illustrated in Figure 1 below.

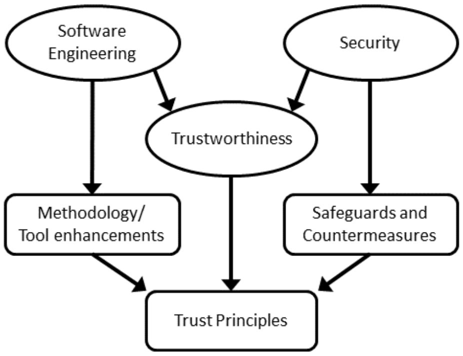
Software Engineering
Security
Trustworthiness
Methodology/ Tool enhancements
Safeguards and Countermeasures
Trust Principles

Fig. 1. Trust Principles that reflect trustworthiness, quality and security of software.

The TSM recognises the role of both security techniques and software engineering practices in the development of trustworthy of software systems, i.e. secure software development can be considered a socio-technical system (STS), which has been examined by Mohammadi et al. [14]. They identified a number of attributes in STS that affect the trustworthiness of the developed software, these attributes are shown in Figure 2.

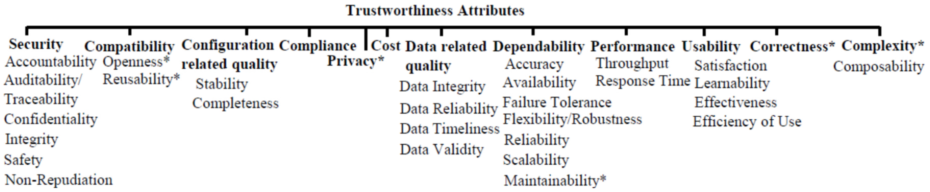

Fig. 2. Trustworthiness attributes identified by Mohammadi et al. [14].

These attributes capture different aspects of trustworthiness, but in this paper we focus on the attributes that are most directly linked with the security of the software product, i.e. Security and Dependability. The difficult part of using these attributes directly, is that it requires a general evaluation of the attributes in all types of software. The metrics created will have to apply to both small and large projects (and everything in between) without being biased toward one or the other.

The work on TSM and STS demonstrate that trustworthy software is defined by directly observable software artefacts of the developed systems, as well as indirectly perceivable characteristics of the software development process. This corresponds to the notions of direct and indirect trust in the trustee.

3.1 Direct Trust

Direct trust can be established through certification (as mentioned above,) through examination of the source code (using different software quality metrics,) or through examination of the security history of the software.

Software related metrics for quality software has been studied extensively in the literature. In the following, we examine a few of these that have been studied in the context of maintainability, but may apply to several of the Dependability related trustworthiness attributes. The maintainability attributes are particularly relevant, because they aim to measure the size and complexity of the developed software and complexity is often consider the mother of all security vulnerabilities.

The simplest way to measure the complexity of software is to measure the size of the program. This is most commonly done by counting the *lines of code* in the software, because smaller programs are often easier to comprehend, but several extensions to this metric are possible. In *effective lines of code*, lines consisting only of comments, blank lines, and lines with standalone brackets are ignored. The *logical lines of codes* counts the number of statements in the programming language, e.g. lines ending with a semicolon in ”Java” or ”C”. The *comment to code ratio* is another simple metric is to measure the amount of actual code to the amount of information included to assist software developers. Other metrics have been proposed, e.g. Halstead complexity [11] tries to eliminate the impact of the programming language by considering the software vocabulary and the program length and the ABC metric [10] is measured as the length of a three-dimensional vector measuring the number of ***A***_ssignments_, ***B***_ranches_, and ***C***_onditions_.

In order to establish the security history of software, it is necessary to identify known vulnerabilities and estimate their severity, i.e. how much control of the system an attacker may obtain and how easy the vulnerability is to exploit in practice. The Common Vulnerabilities and Exposures [21] database maintained by the MITRE Corporation since 1999. It defines a common identifier (the CVE number) and a short description of the vulnerability, which provides security professionals an unambiguous way to discuss vulnerabilities. In adition to naming and describing known vulnerabilities, the Common Vulnerability Scoring System (CVSS) [8] defines a way to capture the principal characteristics of a vulnerability and calculate a numerical score reflecting its severity. The CVSS scoring system is maintained by a special interest group (SIG) of the Forum of Incident Response and Security Teams (FIRST) and CVSS scores for most vulnerabilities listed in the CVE database can be found on the National Vulnerability Database [16]. The CVSS scores vulnerabilities from 0 to 10, where 10 is the most critical. Numerical scores are, however, difficult to communicate to a large audience, so

FIRSThas decided to define severity levels that map the CVSS scores to text. This mapping is shown in Table 1.

Severity level	CVSS Score
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Table 1. Mapping CVSS scores to severity levels

3.2 Indirect Trust

Some software requires frequent patches and security updates, which reflects the competence, development methods and priorities of the development team, i.e. some development teams seem to favour feature rich software delivered at a high rate, while others favour more judicious methods focusing on formal specification, evaluation and testing. Indirect trust may therefore be established through the track record of the software developers. It is hard to know the development methodology of a software product that has not undergone certification, e.g. according to the Common Criteria mentioned above, so we need to rely on inference based on externally observable indicators, such as the number of CVEs and the CVSS scores mentioned above.

Wang et al. [24] proposes a security score based on the CVE and CVSS scores, which are used together with the Common Weakness Enumeration (CWE) [22] to define a security metric. The focus of the security metric is on the common types of vulnerabilities, and how the specific type are handled by the software developers. The calculations use the CVSS score and the duration of the vulnerability type to calculate a severity score of the project.

4 CodeTrust

As mentioned above, the trustworthiness of software depends on both software artefacts, that can be observed directly in the source code, and the methodology, processes and people involved in the software development. In this paper, we develop a metric for the trustworthiness of software called *CodeTrust*. This metric focus on the security aspects of trustworthiness and, in particular, aspects relating to the development process (the software development team) and the security history of a particular software project or product.

4.1 Overview

An overview of the CodeTrust metric is shown in Figure 3, which should be read top down.

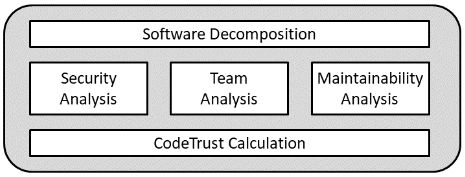

Fig. 3. Overview of the CodeTrust metric.

Most larger software systems, and many smaller systems, include externally sourced software modules or libraries. The first step is therefore to decompose the software into clearly identifiable components. Aggregation of software components is normally achieved through copying source code or through external calls to libraries or services - these are normally known as dependencies. We do not presume to be able to distinguish between internally and externally developed software from the source code alone, so in this paper we propose a decomposition based on the stated dependencies of a given software system or component.

Information about the development team and software source code are not normally available for external review, so the Team Analysis and Maintainability Analysis are only included for internal software projects or Open Source Software (OSS) projects. We believe that evaluating different OSS components to include in current development projects will be one of the principal uses for CodeTrust, which is why we include it in the general metric.

4.2 Security Analysis

The number of CVEs reported for a particular software reflects the number of vulnerabilities found in that software, so a low number of vulnerabilities is always preferred. The criticality of a vulnerability is rated by the CVSS on a 0 - 10 scale, where 10 is a vulnerability with critical severity and 0 has a low severity level. A large number of vulnerabilities with low severity is preferred to a few critical vulnerabilities, as the non-critical vulnerabilities may not provide much access to information, whereas a critical vulnerability may compromise the entire system.

Security Score We calculate a *Security Score (SS)*, where the vulnerabilities and severity and are weighted as seen in equation 1. The Security Score correspond roughly to a classic definition of risk as the likelihood of an unwanted event multiplied by the consequences of that event happening.

$$security\_score = 0.2 \cdot vulnerability\_score + 0.8 \cdot severity\_score \tag{1}$$

Vulnerability Score The *vulnerability_score* indicates the general amount of CVEs registered for the software and thus the developer's ability to create secure software. The *vulnerability_score* calculation is found in Equation 2. The severity score is an indication of how the software matures, by tracking the severity levels of reported vulnerabilities; the calculations are found in equation 6. The weight between the vulnerability score and severity score favours severity, because the criticality of a vulnerability is more important than the amount of vulnerabilities overall.

$$vulnerability\_score = grade\_ncve \cdot grade\_tcve \tag{2}$$

The vulnerability score is made up by the average annual number of recorded vulnerabilities for the software and is represented by the factor *grade_ncve*; the calculations can be found in Equation 3.

$$grade\_ncve = \begin{cases} User\ evaluation & \text{if } ncve <= 5 \\ 0.7 & \text{if } 5 < ncve <= 20 \\ 0.9 & \text{if } 20 < ncve <= 70 \\ 1.0 & \text{if } 70 < ncve \end{cases} \tag{3}$$

The ncve is the average annual number of recorded CVEs, which has a tendency to discriminate against larger projects, because a project with 20 million lines of code is more likely to produce a large number of vulnerabilities compared to a project in the tens of thousands lines of code. This is an issue when CodeTrust evaluation is used as an absolute estimate of software trustworhiness, but it allows comparison of similar software, such as the most popular web browsers Edge, Chrome and Firefox. Possible solutions to the size bias include weighting the *grade_ncve* by the size of the project or separating software projects into size intervals for comparison with other projects of similar size. The *User evaluation* relies on the registered number of users for a particular software project. Such data can be obtained as sales numbers for commercial software or taken from OpenHub [4], which have extensive data on a large set of OSS Projects, including numbers of users and contributors.

$$User\ evaluation = \begin{cases} 0 & \text{if } project\ not\ found \\ 0 & \text{if } 500 < users \\ 0 & \text{if } users < 500 \text{ and } 15 < contributors \\ 10 & \text{if } users < 500 \text{ and } contributors < 15 \end{cases} \tag{4}$$

The second part of the vulnerability score is the CVE trend, which is a simple linear regression of the CVEs for every year. The evaluation does not focus on the amount of CVEs but the trends, which is indicated by the slope in a linear regression.

$$grade\_tcve = \begin{cases} 4 & \text{if } a < -0.2 \\ 7 & \text{if } -0.2 < a < 0.2 \\ 10 & \text{if } 0.2 < a \end{cases} \tag{5}$$

For simplicity, we only operate with a positive trend, i.e. there are fewer vulnerabilities so the slope (a in Equation 5) is negative, a neutral trend (the slope is roughly flat) and a negative trend (the slope is positive). The possible values returned by the calculation have been distributed evenly in the domain 0 - 10, as indicated in the equation above.

Severity Score The *severity score* is based directly on the CVSS reports from the National Vulnerability Database [16].

The severity score, shown in Equation 6, measures whether the average of vulnerabilities with severity *Critical* or *High* is greater than 25% (*average_critical* and *average_high* are binary variables that are true if this is the case.) It also includes the trends, for each of the CVSS severity levels, calculated using Equation 5, in the same way as the CVE trend. The coefficients for the trend variables reflect the relative importance of the severity levels for the final security of the software. They have been established through minor tests with known open source software projects; more fundamental experimentation will be carried out as part of our future work.

$$\begin{aligned} severity\_score = &(0.6 \cdot average\_critical + \\ &0.4 \cdot average\_high) + 0.45 \cdot trend\_critical + \\ &0.3 \cdot trend\_high + 0.1 \cdot trend\_medium + 0.05 \cdot trend\_low \end{aligned} \tag{6}$$

The security score is in the same range (0 - 10) as the CVSS score, with severity 10 meaning the least secure system. We believe this is counter intuitive, so we define the *security_history_score*, in Equation 7, where 10 reflects a system with the best security history.

$$security\_history\_score = 10 - security\_score \tag{7}$$

In the following, we examine the two remaining constituent elements of the CodeTrust metric, namely the team analysis and the maintainability analysis. As mentioned earlier, we focus these efforts on OSS, because numbers to support our analysis are more easily obtained. It should be possible for most software development companies to obtain similar numbers from their own products.

4.3 Team Analysis

The quality of software developed by a development team depends on the project management methodologies, processes and tools employed by the team, but to a higher degree on the security conciousness and experience of the individual team members. The experience of a developer in OSS projects can be roughly estimated by looking at the security history of previous projects. This *contributor score* can be determined as a sum of the *security_history_score* of all of all other projects that the contributor has contributed to, weighted by the relative size of the contributions as shown in Equation 8. This estimation is obviously rough, because the contributor may not have committed any of the vulnerable code to software with a poor security history, but we believe that security conscious programmers will migrate away from projects where security has low priority (in open source projects, the satisfaction of contributing to a meaningful is the primary reward for programmers.)

$$contributor\_score = \sum_{projects} \frac{commits_{contributor\ in\ project} \cdot security\_history\_score_{project}}{total\_commits_{contributor}} \tag{8}$$

Having estimated the experience of the individual contributors to a software project, we can now calculate a team score as the sum of the contributor scores for the individual developers weighted by the relative size of their contributions to the overall software project, as shown in 9 below.

$$team\_score = \sum_{contributors} \frac{contributor\_commits_{project} \cdot contributor\_score}{total\_commits_{project}} \tag{9}$$

The information about contributors to OSS projects and the number of commits that they have contributed to different project can be found on OpenHub [4].

4.4 Maintainability Analysis

As discussed in Section 3, there are many metrics for measuring the maintainability og software. In the current definition of CodeTrust, we favour a simple definition based on the comments to code ration defined in Equation—10, but we clearly identify this as an interesting area for future work.

$$comments\_code\_ratio = \frac{lines_{comments}}{lines_{code} + lines_{comments}} \tag{10}$$

The necessary information for these calculations can be found on OpenHub for many OSS projects. This allows us to benchmark individual projects against the other projects, i.e. normalize the comment to code ration for a particular project to the average project. Table 2 presents data for a number of well known projects; these were collected in December 2016.

The table shows that the majority of OSS projects has a comment to code ration between 10% and 25%, so we define the Maintainability score as the normalized comment to code ratio as defined in Equation 11.

$$Maintainability\_score = \frac{comment\_code\_ratio - 0.1}{0.25 - 0.1} \cdot 10 \tag{11}$$

Project	Lines of code	Lines of comments	comments/code ratio
Apache Subversion	660,711	208,243	24.0%
MySQL	2,862,087	692,663	19.5%
Ubuntu	911,004	187,691	17.1%
Linux Kernel	18,963,973	3,872,008	17.0%
Mozilla Firefox	14,045,424	2,825,225	16.8%
Chromium	14,945,618	2,752,467	15.6%
Python	1,030,242	184890	15.2%
PHP	3,617,916	587,629	14.0%
Git	774,674	96,554	11.1%
Apache HTTP server	1,832,007	210,141	10.3%
neat project	23708	1695	6.67%

Table 2. OpenHub data for popular open source projects.

4.5 CodeTrust Metric

Having calculated the *security history score*, the *team score* and the *maintainability score* for the projects above, we are now able to define the CodeTrust metric as shown in Equation 12.

$$\begin{aligned} codetrust &= 0.65 \cdot security\_history\_score \\ &\quad + 0.25 \cdot team\_score + 0.1 \cdot maintainability\_score \end{aligned} \tag{12}$$

The weighting of the different scores reflect our estimate of the relative importance of these scores to the overall trustworthiness of the developed software. As each of these scores are subject to further research, it is possible that these weights will change, but we feel confident that the relative importance of the different aspects are captured by the weights above.

5 Evaluation

We have developed a simple prototype that implements the CodeTrust metric, which allows us to present a preliminary evaluation. The prototype is written in Java, with the incorporation of a few external components and libraries, such as the Debian Linux package manager program `apt-rdepends` for resolving dependencies and the RoboBrowser [5] for scraping web pages. For a more detailed description of the prototype and the evaluation presented in this paper, please refer to the M.Sc. thesis of Michael B. Nielsen—[15].

We evaluate each of the three components before we present the evaluation of the full CodeTrust prototype.

5.1 Security History Score

We have selected a number of popular OSS projects, where information is available on OpenHub; this is simply to make our evaluation task easier. The results of this evaluation are presented in Table 3

Project	ncve	tcve	vs	lt	mt	ht	ct	ah	ac	sev	ss
Apache server	0.9	10	9	7	10	10	10	0	0	8.85	8.88
Atom editor	-	-	-	-	-	-	-	-	-	-	0
Docker	-	-	-	-	-	-	-	-	-	-	0
Mozilla Filezilla	-	-	-	-	-	-	-	-	-	-	0
Firefox	1	10	10	7	10	10	7	0	1	8.1	8.48
Keepass2	-	-	-	-	-	-	-	-	-	-	10
MongoDB	-	-	-	-	-	-	-	-	-	-	0
MySQL	0.9	10	9	10	10	4	7	0	0	5.85	6.48
neat-project	-	-	-	-	-	-	-	-	-	-	0
OpenSSL	0.7	10	7	7	10	10	10	0	0	8.85	8.48
PHP	1	10	10	4	4	4	4	1	0	4	5.2
Python	0.7	10	7	7	7	4	7	0	0	5.4	5.72
Ruby	0.7	10	7	7	10	7	7	0	0	6.6	6.68
Ruby on Rails	0.7	10	7	7	10	7	7	0	0	6.6	6.68
tar	0.7	10	7	7	7	7	7	0	0	6.3	6.44
Wordpress	1	10	10	10	10	10	7	0	0	7.65	8.12

Table 3. Security History Score for selected OSS projects.

The dashes indicates that there were no CVEs registered for the project at the time of evaluation. This may be explained by projects being new (no vulnerabilities have been found yet), small (few users to find vulnerabilities and little interest from criminals) or well engineered and competently programmed. We note that most of the projects that have no CVE data receives the lowest possible security score, apart from the password manager `Keypass` which receives the highest score. This demonstrates that it is possible for a project to receive a high score based on user evaluations and that projects developed for security purposes are likely to be more security concious, which should result in a higher security score. The relative high score of OpenSSl confirms this belief, despite the bad publicity attracted because of the HeartBleed vulnerability.

5.2 Team Score

The team score is used to describe how the current contributors experience and performance have been with their previous projects. The perfect situation would be to test each contributors performance in their commits, but this is unfortunately not possible. The contributors found are only the contributors currently contributing, which means the contributors with commits during the last 12 months. The contributors are thus evaluated on the projects which they have been part of and committed to. The data from each project is found in Table 4.

We note that two projects (Filezilla and Keepas2) have very few contributors, so if these contributors have not recently contributed to other projects, the team score is essentially determined by the user evaluation of the security history score calculation (this explains the extreme values 0 and 10).

Project	Contributors	Team score
Apache server	27	5.71
Atom editor	822	9.89
Docker	532	9.91
Filezilla	1	10
Firefox	1087	5.14
Keepass2	2	0
MongoDB	106	9.96
MySQL	126	4.62
neat-project	23	9.87
OpenSSL	126	3.02
PHP	170	5.49
Python	47	9.08
Ruby	42	7.29
Ruby on Rails	554	6.29
tar	4	5.63
Wordpress	34	3.49

Table 4. Team Score for selected OSS projects.

5.3 Maintainability Score

The maintainability score simply reflects the code to comment ratio as discussed earlier. The score is shown in Table 5.

Project	Code/Comment ratio	Maintainability score
Apache server	10.29	0.19
Atom editor	8.26	0
Docker	10.93	0.62
Mozilla Filezilla	12.98	2.00
Firefox	16.75	4.5
Keepass2	16.74	4.49
MongoDB	21.29	7.53
MySQL	19.49	6.32
neat-project	6.67	0
OpenSSL	18.20	5.47
PHP	13.97	2.65
Python	15.22	3.48
Ruby	12.18	1.46
Ruby on Rails	16.07	4.04
tar	11.99	1.32
Wordpress	27.49	10

Table 5. Maintainability Score for selected OSS projects.

We observe that two projects (Atom editor and neat-project) receives the lowest score and one project (Wordpress) scores the highest. We explain this by the effects of normalizing the ratio to the range 10% - 25%, i.e. the two lowest scoring projects both have a comment to code ratio below 10% and the highest scoring project has a ratio above the 25%.

5.4 CodeTrust

The results of our evaluation of selected OSS projects are shown in Table 6, which shows the project, the security history score (SH score), the team score, the maintainability score and the CodeTrust score for the project.

Project	SH score	Team score	Maintainability score	CodeTrust score
Apache server	8.88	5.71	0.19	7.22
Atom editor	0	9.89	0	2.47
Docker	0	9.91	0.62	2.5
Filezilla	0	10	2	2.7
Firefox	8.48	5.14	4.5	7.2
Keepass2	10	0	4.49	6.95
MongoDB	0	9.96	7.53	3.24
MySQL	6.48	4.62	6.32	6
neat-project	0	9.87	0	2.47
OpenSSL	8.48	3.02	5.47	6.81
PHP	5.2	5.49	2.65	5.02
Python	5.72	9.08	3.48	6.34
Ruby	6.68	7.29	1.46	6.31
Ruby on Rails	6.68	6.29	4.04	6.32
tar	6.44	5.63	1.32	5.73
Wordpress	8.12	3.49	10	7.15

Table 6. CodeTrust metric for selected OSS projects.

We first note that several projects received a very low score because of a ”0” in the Security History Score. This is how the CodeTrust metric is designed, but it highlights the problem of incomplete information. In most cases, the low security score reflects the fact that no CVEs have been registered for the project.

We also note that the OpenSSl, which contained the bug that caused the HeartBleed vulnerability described in Section 2, receives a respectable score of 6.81, which means that OpenSSL belongs to the upper quartile of the evaluated projects.

Finally, we note that the CodeTrust makes it possible to compare competing frameworks, such as the combination of Python and PHP against Ruby and Ruby on Rails. Both frameworks are relatively mature frameworks for dynamic web content and generally score well, but PHP scores a little lower than the others, which suggests that the Ruby based combination is a little more secure.

6 Conclusions

In this paper, we examined the problem of trustworthiness of software, in particular software developed in Open Source Software projects. Trustworthiness concerns both functional and non-functional requirements for the developed software, but in this paper we focused on the non-functional requirement *security*.

We have defined the CodeTrust metric to measure the trustworthiness of software, through an evaluation og the security history of the software itself and the development team responsible for the software. We also included a rough estimate of the maintainability of the software in our metric, because complexity of code is one of the main causes for insecure software.

We have employed the CodeTrust metric to evaluate a number of OSS projects, including `OpenSSL`, which contained the bug that caused the higly publicized HeartBleed vulnerability. The CodeTrust evaluation gave a high score to `OpenSSL`, which indicates that the CodeTrust metric is robust against short term effects og widely publicized vulnerabilities.

No metric is more reliable than the input it receives, which is also a problem identified in the evaluation of CodeTrust. New software has no security history, so it is possible that CodeTrust should emphasize the team analysis more in this case. This would, however, only solve the problem when the team consists of experienced software developers with a public track record, so software by many student start-ups would not be trusted (it is an open question whether it should).

The problem of missing data is an important issue that we intend to address in future work. There are two main direction that we plan to pursue: acquisition of more data, through more comprehensive web-scraping techniques, and including a confidence score in the metric, which indicates the quality of the data that the evaluation is based on.

The overall result of our evaluation shows that the CodeTrust metric works well for larger and more mature OSS projects, but that there are some problems evaluation new or small projects. The evaluation shows that most of the projects that rated poorly, did so because of missing data, which means that the current prototype cannot be used to make automated decisions about whether to rely on particular software or not. Our evaluation of the two dynamic web-page frameworks `Ruby on Rails` and `PHP` shows that there is a difference in trustworthiness, but also that both frameworks are mature and have a reasonable security track record. The difference may persuade an organisation to choose one framework over the other, which is another reason for evaluating the trustworthiness of software.

Finally, we wish to further examine the relationship between the Security History Score and the Team Score, in particular for software developers that primarily work on a single project. In this case, the CodeTrust evaluation is based almost exclusively on the security history data.

References

1. The heartbleed bug, http://heartbleed.com/
2. Amoroso, E., Taylor, C., Watson, J., Weiss, J.: A process-oriented methodology for assessing and improving software trustworthiness. In: Proceedings of the 2nd ACM Conference on Computer and Communications Security. pp. 39–50 (1994)
3. Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. In: Proceedings of the 3rd IEEE Information Survivability Workshop (2000)
4. Black Duck: Open Hub, https://www.openhub.net/
5. Carp, J.: robobrowser, https://github.com/jmcarp/robobrowser
6. Cerrudo, C.: Why the shellshock bug is worse than heartbleed. MIT Technology Review, September 30, 2014
7. Commission of the European Communities: Information Technology Security Evaluation Criteria (ITSEC): Preliminary Harmonised Criteria
8. Common Vulnerability Scoring System SIG: The Common Vulnerability Scoring System (CVSS), https://www.first.org/cvss/
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, The Internet Engineering Task Force (2008)
10. Fitzpatrick, J.: More c++ gems. chap. Applying the ABC Metric to C, C++, and Java, pp. 245–264. Cambridge University Press, New York, NY, USA (2000), originally published in C++ Report, June 1997
11. Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA (1977)
12. ISO/IEC 15408: Common Criteria for Information Technology Security Evaluation
13. Lampson, B.W.: A note on the confinement problem. Commun. ACM 16(10), 613–615 (1973)
14. Mohammadi, N.G., Sachar Paulus, M.B., Metzger, A., Koennecke, H., Hartenstein, S., Pohl, K.: An analysis of software quality attributes and their contribution to trustworthiness. Closer 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science 3(3), 542–552 (2013)
15. Nielsen, M.B.: Quality and IT Security assessment of Open Source Software projects. M.sc. thesis, DTU Compute, Technical University of Denmark, 2017
16. NIST: National vulnerability database, https://nvd.nist.gov/
17. Pauli, D.: It's 2017 and 200,000 services still have unpatched heartbleeds https://www.theregister.co.uk/2017/01/23/heartbleed_2017/
18. Randell, B.: System structure for software fault tolerance. SIGPLAN Notices 10(6), 437–449 (1975)
19. Seggelmann, R., Tuexen, M., Williams, M.: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, The Internet Engineering Task Force (2012)
20. The Department of Defense (DoD): Trusted Computer System Evaluation Criteria (TCSEC), TCSEC Rainbow Series Library, Orange Book
21. The MITRE Corporation: Common vulnerabilities and exposures, https://cve.mitre.org/
22. The MITRE Corporation: Common Weakness Enumeration (CWE), http://cwe.mitre.org/about/index.html
23. Thompson, K.: Reflections on trusting trust. Communications of the ACM 27(8), 761–763 (1984)
24. Wang, J.A., Wang, H., Guo, M., Xia, M.: Security metrics for software systems. Proceedings of the 47th Annual Southeast Regional Conference (47), 1–6 (2009)