Scalable Trust Establishment with Software Reputation

Sven Bugiel
System Security Lab
TU Darmstadt (CASED)
Darmstadt, Germany
sven.bugiel@trust.cased.de

Lucas Davi
System Security Lab
TU Darmstadt (CASED)
Darmstadt, Germany
lucas.davi@trust.cased.de

Steffen Schulz
System Security Lab
TU Darmstadt (CASED) &
Ruhr-Universität Bochum &
Macquarie University (INSS)
Darmstadt, Germany
steffen.schulz@trust.cased.de

ABSTRACT

Users and administrators are often faced with the choice between different software solutions, sometimes even have to assess the security of complete software systems. With sufficient time and resources, such decisions can be based on extensive testing and review. However, in practice this is often too expensive and time consuming: When a user decides between two alternative software solutions or a verifier should assess the security of a complete software system during remote attestation, such assessments should happen almost in realtime.

In this paper, we present a pragmatic, but highly scalable approach for the trustworthiness assessment of software programs based on their security history. The approach can be used to, e.g. automatically sort programs in an App store by their security record or on top of remote attestation schemes that aim to access the trustworthiness of complex software configurations. We implement our approach for the popular Debian GNU/Linux system, using publicly available information from open-source repositories and vulnerability databases. Our evaluation shows reasonable prediction accuracy for the more vulnerable packets and good accuracy when considering entire system installations.

1. INTRODUCTION

A longstanding problem in computer science is how to assess the trustworthiness of a given (remote) software system, i.e., if it behaves as expected. To make reasonable decisions, common metrics are required along which *trust* is quantifiable and comparable. Example use cases for this problem are when a system administrator/user must decide between alternative implementations of, e.g., a web server or mail client, when a company decides on what programs are acceptable on employees' laptops that intent to connect to the company network, or even when choosing between different available virtual machines to implement desired services in the cloud. In all these cases, we require an objective, comprehensive metric that works efficiently and scales to systems with several hundred interacting software systems.

In practice, there are several methods to measure the trustworthiness of software components. They can be classified as (1) formal verification, (2) evaluation and testing, and/or (3) reputation systems. However, approaches (1) and (2) do generally not meet the required efficiency and scalability due to the intensive manual work involved. In particular, formal verification is used for scenarios with high associated risks or software classified as mission critical, e.g., military applications, train controllers, or satellite firmware. By specifying (semi-)formally the security goals, adversary model, and requirements, software is developed (or adjusted) such that it is verified to achieve these requirements. However, after several years of research, formal verification is still too expensive for most mainstream applications and requires too much time to keep up with the frequent releases of many mainstream software programs. By contrast, methodical evaluation and testing, for instance, according to Common Criteria or by thorough penetration testing, is more practical and frequently used by industry and government organizations. However, the process is still too slow and costly for many applications and, moreover, potentially error-prone due to the incompleteness of the testing/evaluation procedure. Additionally, the testing results are hard to interpret by a common user, who has to choose between different software solutions for his requirements and who wants to evaluate the security of these options. Finally, we are not aware of any reputation-based system for software programs that meets the required scalability and objectivity, although in practice many users and administrators already use subjective experience and recommendations in their decision making processes.

Contribution.

In this paper, we present a pragmatic and lightweight approach to determine the trustworthiness of complex software systems. We extend on previous works on statistical vulnerability prediction to derive the expected trustworthiness of large commodity computer systems and provide a generic, intelligible metric to aid in the assessment and comparison of software security. We implement and evaluate the accuracy of our assessment scheme that can be used locally, to assist in deciding between different software solutions, or to assess the trustworthiness of remote software systems during remote attestation. When assessing the trustworthiness of complete system installations, we can predict their failure rate within error margins of around 10% to 30%.

STC'11, October 17, 2011, Chicago, Illinois, USA.

2. PROBLEM DESCRIPTION

Whether it is for large scale investment decisions, implementation of communication infrastructures or personal use, system integrators and users are regularly confronted with the decision between different software systems. One major criterion for this decision making process is the *trustworthiness* of the program, i.e., how likely it is to "behave as expected" with regards to the security and trust requirements of the system integrator or user.

However, increasingly complex and rapidly evolving software systems impede efficient security analysis and testing, let alone formal analysis. Instead, users and system administrators typically rely on a vaguely defined reputation system, built from personal experience, recommendations and additional meta-data such as the programming language or development process. This is particularly problematic in cases where combinations of software systems must be evaluated in realtime, e.g., when evaluating client platforms in remote attestation protocols.

We know of no works that investigate the accuracy of these informal assessments. Nevertheless, we expect their accuracy to decrease significantly with lower expertise of evaluator and time available to make the assessment. In the following sections we thus describe and evaluate an automated generic mechanism to derive the likelihood of security failures in a given software, with the goal to support a more accurate decision making processes. We identify the following requirements for a practical and efficient assessment of software trustworthiness.

Accuracy. The produced trustworthiness assessments must be reasonably accurate to support the decision process. Error margins must be available to judge the accuracy of the assessment.

Universality. The desired assessment system must be applicable to any software, i.e., it must not depend on individual characteristics such as the programming language or development process. However, individual characteristics can be used to increase the prediction accuracy.

Objectivity. The system must yield objective and reproducible trustworthiness assessments. In particular, it should be intelligible for the software developers which events or design decisions result in worse trustworthiness assessments and how assessment results can be improved.

Scalability. The derivation of the trustworthiness assessment should be automated to scale with the huge number of software programs available for commodity computer systems. Furthermore, the employed trustworthiness metric should allow efficient computation of the aggregated trustworthiness for combinations of programs based on their individual assessments.

3. SOFTWARE TRUST AND REPUTATION

In the following we present a pragmatic and scalable approach to determine the trustworthiness of software systems through software reputation. We first take a look at how system administrators and users decide for or against using (trusting) a particular software today. Using this perspective, we define a pragmatic, objective notion of *security failure* and *trustworthiness* and then present a metric to express *trustworthiness prediction*.

3.1 State-Of-The-Art in Practical Systems

As lined out in Section 1, our daily decisions for trusting or not trusting a particular software are usually not determined by objective and comprehensive testing and evaluation or even formal analysis, as these approaches are often simply too expensive and time consuming, and scale poorly in today's rapidly changing software landscape.

In practice, the reliability of a program is often derived based on subjective impressions, recommendations and past experience. This reputation-based approach is well-known and in fact very common for human interaction. In the software landscape, such reputation is established based on personal experience with the software, reviews and recommendations by (often only partly) trusted third parties. Such events are in general difficult to track and often require the users to contribute to extensive reputation infrastructures. However, in software security, the overwhelming amount of security incidents are, in fact, objective software defects that are recorded and monitored in public databases. Moreover, multiple previous works show that one can indeed estimate the amount of future security failures based on past experience [12, 14, 13, 21], meaning security failure rate of a program is a relatively stable, characteristic value.

3.2 An Objective Definition of Trust

In general, the definition of *security failure* depends on the security goals and software environment of the respective usage scenarios, and is thus a highly subjective matter. However, from the perspective of the software developer, a different security notion can be identified where a given security failure is either caused by a particular software or not, and a decision is made whether the program should be fixed and users be warned on potentially unexpected behavior. This decision is usually straightforward and determined by common practices in software and security engineering, such as reducing information flow, encapsulating complexity and minimizing unexpected behavior. In this case, the security failure is caused by an *inherent* security flaw in the software. Hence, we can objectively define a software security failure as follows:

Software Security Failure: *A software security failure describes the event where the software's author or the security community decides that a flaw exists in a program that must be fixed to prevent future security incidents when using the program.*

By this definition, an event is classified as security failure based on a *consensus* decision. Individuals (or other communities) may consider this decision as false, however, in practice these decisions are usually straightforward and only few incidents required a more detailed debate, as evidenced by fact that major vulnerability databases and mailing lists also always provide the name and version of the affected software database[1] However, we emphasize that by this definition, a security failure does not imply that an actual attack has succeeded in practice but only marks the *discovery of a vulnerability*, which may lead to an attack under certain circumstances. Based on this definition, we can define

[1]See, e.g., the CVE database by MITRE (`cve.mitre.org/`) or the bugtraq mailing list (`seclists.org/bugtraq/`).

software trustworthiness as the likelihood of future security failures:

Software Trustworthiness: *Software Trustworthiness describes the probability of a particular piece of software to elicit a security failure within a certain time frame.*

Note that this notion of trustworthiness describes a prediction into the future and thus only an estimate with a certain accuracy. To derive this estimate, we use the previously observed effect that the rate of software security failures is rather constant for individual software programs and its modules. For example, two thirds of the Red Hat Linux software packages have been correctly predicted to be vulnerable, with a false positive rate of 17% [13].

3.3 A Metric for Software Reputation

Our system also requires an objective metric to express the trustworthiness of one software as opposed to another. Ideally, the metric should assign absolute trustworthiness values to individual software programs, such that the relative advantage of one program over another can be quantified. Moreover, for optimal scalability it should be possible to combine the assessments of individual software components into assessments for different compositions of similar software systems, as they are most commonly found in to days commodity computing systems.

We meet these goals by basing our trustworthiness metric on the Mean Time Between Failures (MTBF), a well-known and common approach for predicting the reliability of complex systems in electrical engineering [11, 7]. Specifically, since the publication date of vulnerability fixes is often more dependent on the IT support or distributor and thus not actually an objective characteristic of the respective software, we assume a Mean Time To Recover (MTTR) of zero. As a result, since $MeanTimeToFailure(MTTF) = MTBF + MTTR = MTBF$, we effectively measure the trustworthiness of software as MTTF[2].

Figure 1 illustrates the derivation and relationships of MTBF, MTTR and MTTF: The MTBF is calculated as the average of the durations that a component is working (b_i), while the MTTR is the average over the recovery times (r_i). The MTTF is the time between failures ($d_i = r_i + b_i$), and identical to MTBF for $r_i = 0$. To account for changes in authorship and software development practices, we further introduce Weighted Mean Time To Failure (MTTF_λ) as a weighted version of the MTTF where the impact of older security failures decreases exponentially relative to an aging parameter λ:

$$MTTF = \frac{1}{i}\sum_i d_i \qquad MTTF_\lambda = \frac{\sum_i e^{-t_i/\lambda} \cdot d_i}{\sum_i e^{-t_i/\lambda}}$$

To derive the aggregated failure rate of combined software systems, we currently assume that all components depend on each other (worst case scenario). As a result, each component failure also leads to a system security failure, i.e., the failure rates of individual components can simply be added up to estimate the failure rate of an aggregated software system. The assumption appears reasonable considering the lack of strong isolation and information flow control in commodity computer systems. Where strong isolation is available, the assessment can be scoped to derive the trustworthiness of individual software stacks. For example, to judge the security of an imminent security-sensitive transaction on a Linux system, one may argue that software that is installed but not started since boot-up can be ignored.

[2]In engineering, the MTTF is reserved for items that are not repaired but immediately replaced. The semantic can be adopted here as well, since vendors often only inform their customers of security issues once the fix is available, allowing them to replace the software.

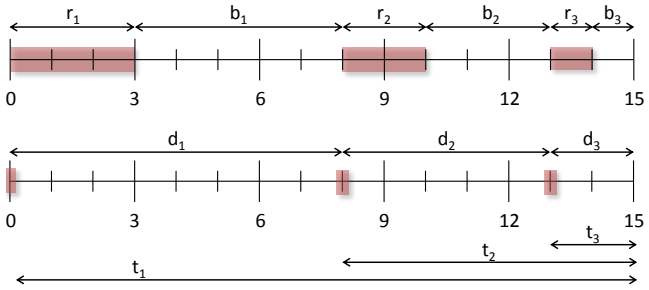

Figure 1: Example for computing MTBF, MTTR, MTTF and MTTF_λ metrics. The upper time line considers recovery periods r_i, while the lower illustrates our case with an assumed MTTR of zero.

In practice, users and administrators are often interested in the estimated amount of security failures for a concrete time period t. For example, when selecting software for a secure isolated online banking compartment using virtualization technology, it may be sufficient to know the security failure rate for that compartment between the enforced system updates. We define this metric as the Secure Transaction Probability (STP), which is derived using the MTTF and time frame t:

$$STP(t) = \frac{t}{MTTF} \qquad STP_\lambda(t) = \frac{t}{MTTF_\lambda}$$

The STP is a function of the time t that the software at the prover must be secure in order for some operation (i.e., a program executing in time t) to be considered secure. For example, if the Mean Time To Failure is twice the transaction time t, then the transaction security is endangered half of the time: $STP(t) = \frac{1}{2}$. Put differently, there will be a security incident between approximately every second security update, meaning half the critical user transactions are in danger of compromise.

4. DESIGN AND IMPLEMENTATION

In this section we design and implement a prototype for computing the trustworthiness of program suites (packages) in a scalable fashion. We do this using the public repositories and package meta-data from Linux open-source software distributions. We approximate software trustworthiness based on public consensus, as by our definition in Section 3.2, by mining the publicly available security advisory archive of the Debian GNU/Linux distribution and the Common

Vulnerabilities and Exposures (CVE) database provided by MITRE and associating the recorded (objective) security failures with individual software packages.

4.1 Architecture

Figure 2 illustrates the overall process of linking package identities to security records for the remote attestation use-case (e.g., using property-based attestation [16]. At the prover, a TCG-style binary measurement list is created directly from program execution flow, for example using Integrity Measurement Architecture (IMA) [17]. The measurements are resolved to their respective binary package identities using publicly available binary package repositories. Based on package repository meta-data, the binary packages are then resolved to source packages, i.e., more abstract versions of the binary packages that are independent of the local system's architecture and binary versions, which are provided to the verifier. At the verifier, the distributor's security advisory archive and third party vulnerability databases are used to associate recorded security incidents with individual source packages. The resulting database of security incidents per source package is then used to derive the trustworthiness metrics as described in Section 3.3. The trustworthiness of the prover is then simply derived by looking up the individual reported source packages in the trustworthiness database of the verifier.

Note that several different configurations are possible depending on the trust model and usage scenario. For example, instead of using TCG-style binary measurements of files, the TCB of the prover might directly record and report the list of installed applications (binary packages) to the verifier, who should then additionally resolve these to source packages. Alternatively, when using the system only for informational purposes, e.g., for ordering programs in an App store by their estimated trustworthiness, only the components on the verifier side are needed.

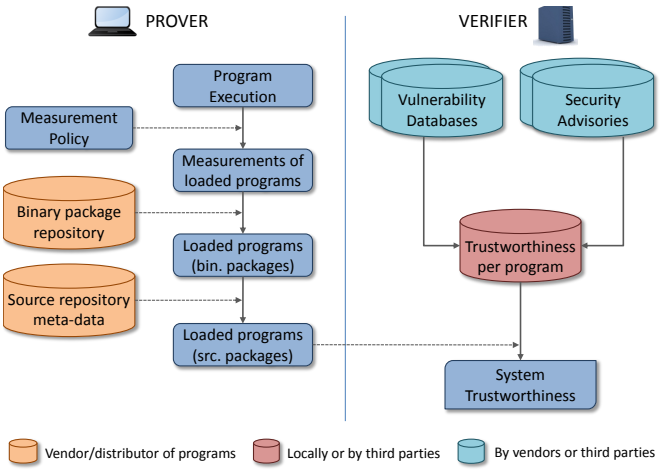

Figure 2: Evaluation of the generic system trustworthiness in systems with package repository.

To fully automate this process, we require the following features:

Program Identifier: To associate trustworthiness ratings or security incident reports of different security information providers and track them through different versions of a program, every program must have a unique identity. The chosen identifiers should be globally unique or mappings between providers must be maintained.

Program Repository: To map the trustworthiness data associated with program identifiers to binary measurements of individual program files, a repository of all versions of deployable binaries and the meta-data to associate this binary data with the abstract program identifier are required.

Advisory Repository: For scalability, we require that the security advisories are readily available and in a consistent format, suitable for automated parsing. An advisory must state which programs are affected, such that the advisory can be mapped to a set of packages in the repository, and which vulnerabilities have been addressed by the update.

Completeness: For accurate trustworthiness estimations and correct reconstruction of each program's history, we require that the advisory and program repositories are complete, i.e., that no vulnerabilities have been silently patched without issuing an advisory, all affected programs are named in an advisory, and all deployable programs are available in the program repository.

Vulnerability Info (optional): For more detailed statistics on the history of vulnerabilities of particular programs, it is useful if the issued security advisories also contain (references to) vulnerability details such as their severity and publication date.

The first three requirements are realistic when considering the well-maintained public software repositories of large open-source software distributors. The completeness requirement for the distributor's security advisory archive is rather strong. However, from our experience large distributors are very aware of this responsibility. The main open-source distributions maintain an internal network for exchange of vulnerability information and the security teams and vulnerability databases follow a policy of full disclosure, allowing developers, users and researchers to verify if vulnerabilities were handled correctly. Public security advisory databases, such as the CVE, provide vulnerability reports in a consistent and parsable manner, including references to severity ratings by NIST.

4.2 Security in mainstream OSS distributions

We reviewed several mainstream Linux distributions as to whether they fulfill the requirements formulated in the previous section. The results are summarized in Table 1.

As expected, all major distributions publish repository information and security advisories, although the commercial RedHat distribution makes the package repositories and package metadata only available to their customers.

All distributors also use the CVE database to refer to individual public vulnerabilities, but with sometimes lacking consistency. The SuSE advisories often describe vulnerabilities in several different packages at once and are not easy to parse automatically. The Fedora advisories also lack structure and lack consistent referencing of the respective CVE records.

Requirement	Debian	Ubuntu	RedHat	Fedora	SuSE	FreeBSD
Program Repository	✓	✓	(✓)	✓	✓	✓
Advisory Repository	✓	✓	(✓)	-	-	✓
Adv. Completeness	✓	✓	(✓)	-	✓	-

Table 1: Fulfillment of requirements by the major Linux distributions

Fedora and FreeBSD fail the completeness requirement. FreeBSD provides security advisories for the core system applications, but apparently follows a much more relaxed policy regarding the several thousand other packages in the ports collection. Fedora mixes security advisories with general update information from their bug-tracking system, which is not easily filtered. In contrast, Debian features an online security bug tracker where the status of incidents can be searched by package name, vulnerability ID or advisory. Only Debian also publishes a list of CVEs that have been reviewed but do not affect Debian, assisting in verifying the completeness of the process. The largely-based Debian system Ubuntu also fulfills our requirements but suffers from rather high frequency of regression bugs due to incorrect security fixes, which would likely distort the derivation of the actual security incident rate per package.

4.3 Introducing TrustMiner

We implemented a prototype named *TrustMiner* for the Debian GNU/Linux distribution. TrustMiner maintains a track record of vulnerabilities per program and computes the software trustworthiness of arbitrary sets of programs as MTTF or MTTF_λ. For attestation, it resolves binary measurements recorded by IMA [17] to the corresponding Debian binary packages, or directly uses a list of binary packages. The binary packages are resolved to source packages using the Debian source repository metadata. On the verification side, TrustMiner uses the CVE vulnerability database, the Debian Security Advisories[3], and the NIST Common Vulnerability Scoring System (CVSS) [10] to create a track record of security incidents for each source package. The list of identified programs running at the prover are matched against the verifier's database of security track records, yielding the overall trustworthiness of the prover's system.

The mechanism can also be queried locally, to aid system administrators in comparing the security of alternative program implementations more objectively. The latter is particularly interesting as Debian also features an *alternatives* framework, where package meta-data is used to declare that a particular package is an alternative implementation of another package, so that apt-sec can be used to provide alternative suggestions with a higher level of trustworthiness upon installation of packages.

Although we did not encounter any fundamental problems which make our approach outright impractical or less scalable, we found multiple minor issues that could be optimized for more effective and accurate verification of remote Debian installations:

- We found no vendor that directly supports the required unique program IDs. Although all the investigated distributions maintain extensive meta-data and source package information, the names are occasionally changed due to political issues. A recent example is the OpenOffice.org suite of packages, which was renamed to LibreOffice as the software project was forked. We decided to maintain a list of source package aliases to take these instances into account.

- Debian and Ubuntu sometimes publish revisions to their advisories due to incorrect and incomplete fixes, or fixes that break other (non-security) functionality. Unfortunately, neither Debian nor Ubuntu appear to have strict policies on how security-critical advisory revisions are distinguished from non-critical regressions. We decided to simply count only the vulnerabilities that have an CVE associated with them. This usually means that distribution-specific security issues are ignored.

- If multiple binary packages are created from a single source package, it is typically unclear which of the binary packages are affected by a flaw in the source package. For simplicity, the distributor will typically redistribute all associated binary packages, resulting in reduced trustworthiness of possibly unaffected programs. For example, the ISC BIND program also includes simple tools for DNS name resolution, which are presumably not affected by all the security flaws of the BIND server.

- Programs using Python or Java usually only ship source code in their binary packages that is compiled on-demand by the respective interpreter or Just-in-Time (JIT) compiler. In these cases, the source files are often not loaded at all during execution, but only their time stamp is checked to see if they should be recompiled. Hence, IMA is not sufficient as a measurement agent but instead the respective JIT compilers and interpreters should be modified to provide the required measurements. However, this limitation is out of scope of this work.

5. EVALUATION

We evaluate our approach using our prototype for the Debian GNU/Linux distribution. We predict the security failure rates for individual packages for the year 2010, using only data available up until 2009. We then compare the actual vulnerability records of 2010 with our estimations to find the accuracy that can be expected from our approach. Finally, we do the same evaluation for complete system installations, represented by three typically configurations.

5.1 Base Data

We used the Debian Security Advisorys (DSAs) published as of 2001, starting with "DSA-011", as it is the first DSA with a format suitable for automatic parsing. The last considered DSA is "DSA-2139", resulting in 2129 considered DSAs and 4530 unique vulnerabilities in 443 programs. Figure 3 shows the overall distribution of actual vulnerabilities recorded per package in this period. We show only the name of every fifth package due to space constraints.

[3] `http://www.debian.org/security/`

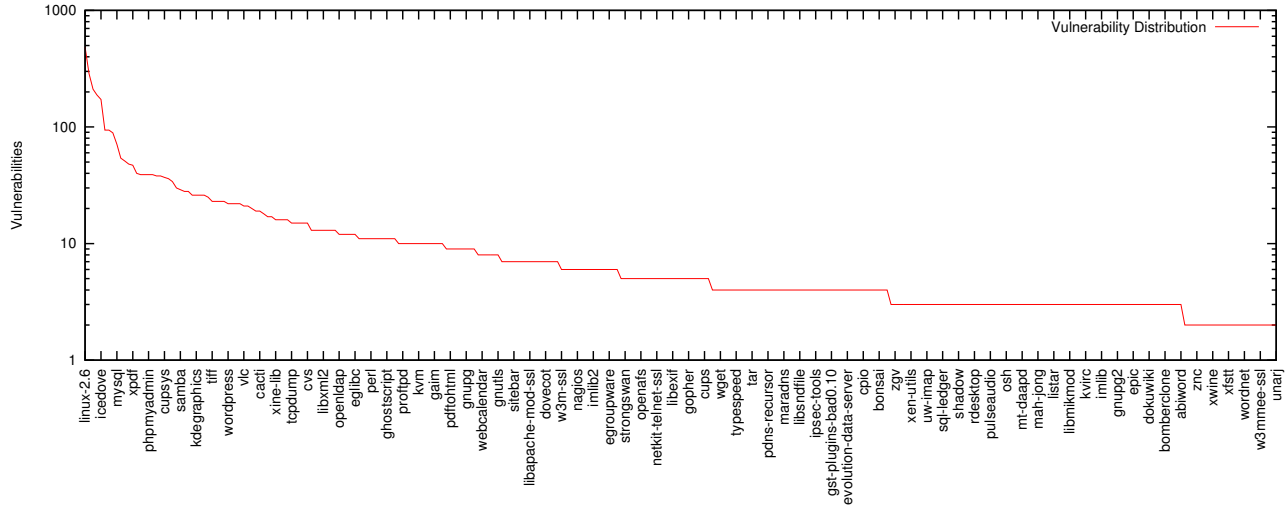

Figure 3: Number of vulnerabilities per program, sorted to illustrate the distribution of vulnerabilities. (Note that only every fifth program name is displayed on the x-axis.)

Table 2 shows the 20 packages with the most recorded vulnerabilities until 2010, their respective MTTF for $\lambda = 12$ and the corresponding estimated ($STP_{12}(365)$) and actual (Vuln. 2010) amount of vulnerabilities for 2010. Note that for licensing reasons, Debian lists the Mozilla Firefox, Thunderbird and SeaMonkey programs as iceweasel, icedove, and iceape. Readers unfamiliar with Debian Linux may lookup the other package names in the Debian package directory `http://www.debian.org/distrib/packages`.

Considering the distribution of vulnerabilities shown in Figure 3, it is clear that vulnerabilities are not equally distributed but clearly follow a pattern. This is consistent with the results of related work such as [14], and substantiates our working hypothesis that every program has a certain characteristic security failure rate.

It is also interesting to note that the Linux kernel has the by far highest rate of security bugs. This confirms the common knowledge between Linux system administrators that it is very hard to isolate local users and prevent them from gaining root privileges, which is particularly problematic when using it in systems like Xen or Android that cannot keep up with the rapid development and (often silent) patching of potential security bugs.

Vuln. < 2010	$MTTF_{12}$	$STP_{12}(365)$	Vuln. 2010	Packet
391	3.1	118.9	89	linux-2.6
288	169.2	2.2	-	linux-2.4
188	17.6	20.7	23	iceweasel
171	14.4	25.3	23	icedove
158	6.2	59.2	53	xulrunner
94	351.4	1.0	-	mozilla
90	274.0	1.3	8	php
89	28.1	13.0	22	iceape
83	38.1	9.6	10	wireshark
56	552.3	0.7	15	mysql
51	73.1	5.0	-	clamav
45	42.3	8.6	2	xpdf
40	58.6	6.2	0	ruby
40	73.9	4.9	7	cups
38	205.6	1.8	0	xorg-server
35	118.8	3.1	4	openssl
35	132.5	2.8	3	krb5
33	105.8	3.5	6	postgresql
31	62.4	5.9	8	phpmyadmin
30	533.6	0.7	0	mantis

Table 2: Number of vulnerabilities and MTTF of the top 20 vulnerable packets until 2009, and the respective vulnerability predictions (STP) and actual vulnerabilities recorded in 2010. Discontinued programs are marked with "-".

5.2 Prediction Accuracy

We use the software trustworthiness to compute the number of expected future vulnerabilities per year as $STP_{12}(365) = 365\text{days} \cdot MTTF_{12}$. Figure 4 shows the *predicted* vulnerabilities per package for 2010, as well as the corresponding *actual* vulnerabilities per package that have been recorded in that year. We removed all packages with a prediction rate of less than 1 vulnerability per year, leaving 196 packages.

Sorted again by the number of vulnerabilities per package, the correlation between prediction and observation is clearly visible. However, it also becomes clear that the prediction accuracy decreases with reduced overall security failure rate.

To compare TrustMiner to existing vulnerability prediction schemes, we calculate the average recall and precision as used in previous works [14, 13, 21]. However, previous approaches only use a classification of packages as vulnerable or not using Support Vector Machines (SVMs) while our approach focuses on finding a useful absolute estimate for the number of vulnerabilities to expect. For the purpose of this comparison we thus simply treat every package with an $STP_{12}(365) \geq 2$ as vulnerable. This results in an average recall of 54%, meaning that of all vulnerable packages, 54% are marked as vulnerable. Furthermore, of all the packages TrustMiner marked as vulnerable, 57% are actually vulnerable (precision). As can be seen in Table 3, these results are comparable to [14] (predicting vulnerable components of

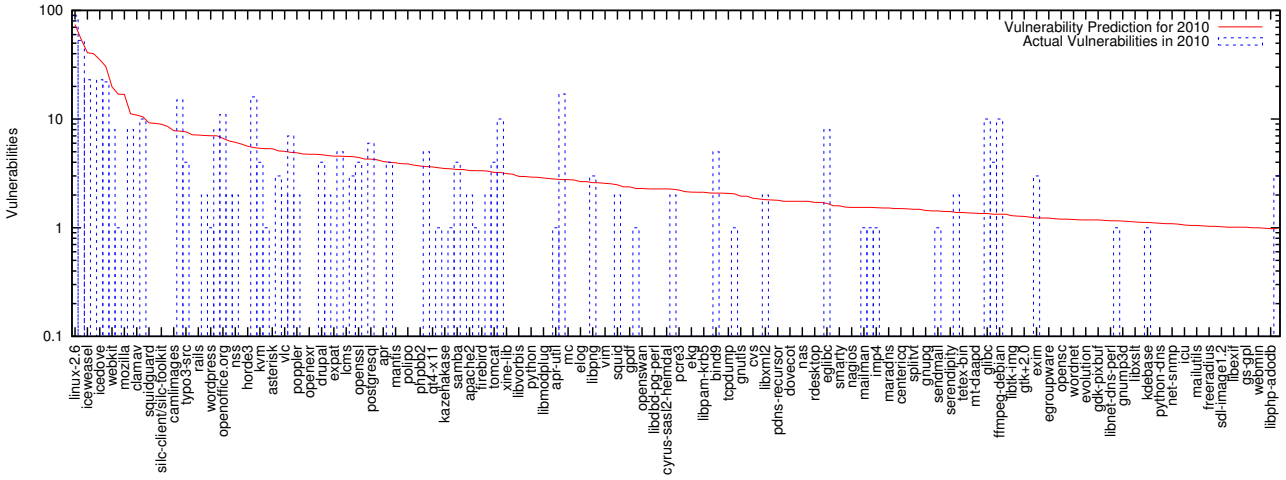

Figure 4: Number of predicted (solid line) and actual vulnerabilities (dashed boxes) for 2010, sorted by prediction rate. Only every second package name is displayed.

the Firefox Browser), and slightly more accurate than those of [21] (predicting vulnerable programs in Windows Vista). However, TrustMiner does not provide the same accuracy as the SVM-based approach presented in [13], which predicts the vulnerability of packages using packet dependencies. While the authors of [13] are not completely sure why this metric provides such high accuracy, we hope to combine it with our system in the future to provide maximum prediction accuracy. However, we aim to maintain our advantage of having a metric that can easily be understood by humans and can also provide a notion of the relative trustworthiness distance between two alternative software implementations.

	TrustMiner	[14]	[13]	[21]
Recall	54%	45%	65%	20%
Precision	57%	70%	83%	66%

Table 3: Predication Accuracy of TrustMiner vs. other approaches. Of all vulnerable packages, 54% **are marked as vulnerable. Of all packages marked as vulnerable,** 57% **actually were found to be vulnerable.**

The rate of security incidents and thus also the accuracy of our prediction fluctuates strongly for packages relatively very low security incident rates. As also observed in previous works, the accuracy increases if the package is more vulnerable. As a result, our system suffers low accuracy for a very large amount of packets, resulting in rather bad overall precision and recall rates. However, the relation between security incidents and prediction accuracy also means that the overwhelming amount of security incidents are predicted rather accurately as they correctly estimated vulnerability rates form the bulk of the overall security incident rate.

Even for some programs with a larger amount of vulnerabilities per year, strong fluctuations are visible. This is likely due to the changing interest of penetration testers and vulnerability researchers and also due to the fast development and replacement of software in current Linux desktop systems. As a result, the vulnerability rates of individual packages randomly increase for shorter periods of time, resulting in the strong noise visible in Figure 4. However, such fluctuations should cancel each other out when considering systems consisting of several software packages. As a last step, we thus consider the prediction accuracy of aggregated systems instead of only individual software packages.

Evaluation of Aggregate Software Systems.

To derive the prediction accuracy for entire system installations, we evaluate three typical Debian installations as well as the Debian base installation. This process is very similar to assigning an overall trustworthiness value during remote attestation of the respective system. However, as explained earlier, our prototype can also use the packages list provided by the installer instead of binary measurements provided by IMA. We use this feature here to attest not only the executed applications but the complete systems. Specifically, we consider the following three installation types:

- The base system represents the minimal Debian installation and yields the highest (but still rather low) trustworthiness.
- The desktop system is comprised of the K Desktop Environment (KDE) and X11, the Mozilla browser and email clients Firefox and Thunderbird, OpenOffice and the VLC video player.
- The server system is a Linux/Apache/MySQL/PHP (LAMP) installation with an additional Lightweight Directory Access Protocol (LDAP) server and OpenSSH.

Table 4 shows the overall MTTF and MTTF_{12} for the respective systems as well as the corresponding expected and actual vulnerability rates for the year 2010. The achieved prediction accuracy is significantly higher than for the above consideration of individual programs. The MTTF metric performs better for the server and base system, while the MTTF_{12} metric suffers less in the Desktop installation. We assume this is due to the higher stability of the server and base system components over time. Current Linux desktops and desktop applications get developed, forked and replaced rapidly, resulting in short-lived applications with rather high vulnerability rates.

Vulnerability rates	Base System	Desktop	Server
MTTF (days/vuln.)	3.6	1.1	2.8
$MTTF_{12}$ (days/vuln.)	2.7	1.1	2.4
STP(365) (vuln./yr)	101.5	322.6	128.8
STP_{12}(365) (vuln./yr)	134.7	301.7	153.4
Actual Vuln. in 2010	106	222	116
MTTF Accuracy	95.6%	68.8%	90.1%
$MTTF_{12}$ Accuracy	78.7%	73.6%	75.6%

Table 4: Expected and actual vulnerability rates in 2010 for different system setups.

6. RELATED WORK

6.1 Remote Attestation Concepts

Remote attestation aims at establishing trust into a remote computer systems through verifiable reporting the system's software state. The original concept introduced by the Trusted Computing Group [19, 17] uses cryptographic hashes of the software binaries to allow the identification of installed software on the reporting system. Extensions to this design were proposed that use virtualization technology [3, 4], mandatory access control schemes [6], or isolation mechanisms in hardware [9, 8, 18] to minimize the attested code base. Although these schemes improve the efficiency and scalability of remote attestation, they still only report the identity of software components but not their security or trustworthiness.

The concept of property-based attestation (PBA) was introduced to address this problem [16, 15, 2]. Their idea is to bind the identity of individual programs to certain security properties using digital signatures. However, these works do not specify what kinds of properties to report or how to measure them.

In this context, our approach can be seen as a possible extension of existing remote attestation schemes that considers specifically the validation of reported software configurations at the verifier. While our prototype uses IMA measurements as input, one might also use our predicted trustworthiness values per package in context of PBA, as one of several properties to attest to.

6.2 Vulnerability Metrics and Prediction

As we described in Section 4, our approach is based on traditional software reliability measurements such as the MTTF. These metrics have been developed in prior works [7, 11], however, they mainly target software functionality instead of software security. Regarding security metrics, a generic software security metric based on the CVSS scoring system is presented in [20]. They consider the severity and risk of the vulnerabilities (provided by the CVSS scoring system) to derive the security of a particular software. However, they do not substantiate the motivation of their security metric definition. Further, they only apply their security metric to selected web browser and webserver applications.

Multiple prior works consider the prediction of vulnerabilities using Support Vector Machines (SVMs). In [21] the authors analyze vulnerability prediction for Windows Vista, finding high correlation between metrics such as code churn, complexity, coverage and organizational measures and the amount of vulnerabilities. They deploy logical regression methods to predict future vulnerabilities, but suffer from a rather low recall, of about 20%. Vulture [14] analyzes vulnerability databases and archives to associate past vulnerabilities with specific program components in Mozilla Firefox. Using SVMs. Their recall and precision rates are similar to TrustMiner (cf. Section 5.2, however, Vulture requires access to source code and manual intervention to associate vulnerabilities with the right source code components, reducing scalability. Another work found a strong correlation between vulnerabilities and package dependencies [13]. They applied SVMs on dependency data and vulnerability reports to predict vulnerabilities in RedHat Linux. They report rather good prediction accuracy, predicting two third of vulnerable packages (65% average recall) and correctly marking 4 out of 5 packets as vulnerable (83% precision). The authors of [1] use the vulnerability density, which reflects the number of vulnerabilities in a given size of code, and the vulnerability discovery rate to model and determine the amount of hidden vulnerabilities in Windows XP and RedHat Linux.

Our approach differs from prior work in our choice of the metric, which is very intuitive and provides richer semantics for subsequent security assessment than the classification approach using SVMs. Specifically, our metric allows to directly derive the security of aggregated systems and also to quantify the relative difference in trustworthiness between any two packages. Yet, we achieve similar accuracy rates, especially when considering entire system installations as shown in Section 5.2.

7. LIMITATIONS AND FUTURE WORK

We presented a pragmatic approach to measure the trustworthiness of a complex software system. Our proposal is based on software reputation schemes in the sense that we derive the trustworthiness of individual software packages from the number and timing of reported vulnerabilities.

Although our prototype makes some strong assumptions on the availability and consistency of the used data, these are easily met by larger free software distributors such as Debian and Ubuntu that feature public software repositories and comprehensive security support. However, all of the major Linux distributions we examined could improve the accessibility of their security advisory databases by making them more consistent and machine readable.

By leveraging the information provided in the CVE and advisory databases, we essentially rely on public consensus to derive our security statistics and, hence, assessments. It is certainly possible that the used databases contain incomplete or wrong information. In particular, the security team of the distributor may misjudge the impact of vulnerabilities on certain packets or simply miss them. However, both databases are subject to public scrutiny and the past has shown that any problems in, e.g., Debian's handling of security issues, receives widespread attention and analysis [5]. Hence, it is unlikely that sufficient misinformation accumulates to significantly bias the statistical results.

For future work, it would be interesting to see if the severity ratings according to the CVSS also follow a pattern that can be used to increase prediction accuracy, and if the approach of [13] using package dependencies can be adopted to improve the accuracy of our system. Based on available package meta-data in Debian, we aim to extend TrustMiner to provide a tool that automatically suggests alternative (more secure) implementations of similar applications.

8. REFERENCES

[1] O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[2] L. Chen, H. Löhr, M. Manulis, and A.-R. Sadeghi. Property-based attestation without a trusted third party. In *Information Security Conference (ISC)*. Springer, 2008.

[3] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2003.

[4] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, 2004.

[5] K. N. Jacob Appelbaum, Dino Dai Zovi. Crippling crypto: The Debian OpenSSL debacle. In: The Last Hackers On Planet Earth (HOPE), 2008.

[6] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-reduced integrity measurement architecture. In *ACM Symposium on Access control models and technologies (SACMAT)*. ACM, 2006.

[7] M. R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., 1996.

[8] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: efficient TCB reduction and attestation. In *IEEE Symposium on Research in Security and Privacy (S&P)*. IEEE, 2010.

[9] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*. ACM, 2008.

[10] P. Mell, K. Scarfone, and S. Romanosky. A complete guide to the Common Vulnerability Scoring System Version 2.0. `http://www.first.org/cvss/cvss-guide.pdf`, 2007.

[11] J. D. Musa, A. Iannino, and K. Okumoto. *Software reliability: measurement, prediction, application*. McGraw-Hill, Inc., 1987.

[12] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *International conference on Software engineering (ICSE)*. ACM, 2006.

[13] S. Neuhaus and T. Zimmermann. The beauty and the beast: vulnerabilities in RedHat's packages. In *USENIX Annual Technical Conference (USENIX)*. USENIX, 2009.

[14] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Conference on Computer and Communications Security (CCS)*. ACM, 2007.

[15] J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner. Property attestation — scalable and privacy-friendly security assessment of peer computers. Technical Report RZ 3548, IBM Research, 2004.

[16] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *New Security Paradigms Workshop (NSPW)*. ACM, 2004.

[17] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*. USENIX, 2004.

[18] E. Shi, A. Perrig, and L. van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Research in Security and Privacy (S&P)*. IEEE, 2005.

[19] Trusted Computing Group (TCG). *TPM Main Specification, Version 1.2, Revision 116*, 2011.

[20] J. A. Wang, H. Wang, M. Guo, and M. Xia. Security metrics for software systems. In *Annual Southeast Regional Conference*. ACM, 2009.

[21] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010.