The Mathematics of Obscurity: On the Trustworthiness of Open Source

Hermann Härtig, Claude-Joachim Hamann, and Michael Roitzsch
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
{haertig,hamann,mroi}@os.inf.tu-dresden.de

ABSTRACT

It is more difficult to find errors when source code is secret. More people search for errors when source code is public. These counteracting effects are pivotal to the question whether openness fosters security. Errors in software are found by people with either constructive contribution or exploitation in mind. Focusing exclusively on this discovery aspect, we present a probabilistic model, which allows us to compare the open source and closed source situations.

We start out with our assumptions explained using a simple introductory model. We then extend this to what we believe to be an adequate model of a bug-hunting process conducted by multiple competing parties. The model employs an asymmetric race paradigm. One of the surprising results is that even an arbitrarily large group with good intentions cannot safely dominate the evil attackers. Instead, they are limited by a significant upper bound in their winning chances.

1. INTRODUCTION

What is more secure, software with secret source code or software with open source code? — If you ask a randomly chosen group of computer-literate people this question, prepare to find yourself in the middle of a brawl on whose opinion is correct.

One half of the group will argue pro open source, claiming that the entire world becomes one united bug-hunting army, because of everyone having access to the source code. And since anyone can use the source and compile the executable himself, anyone can now contribute patches to help improve the software. This means that bugs will be found quickly, fixes are developed fast and deployment is instantaneous, because everyone immediately benefits from the published corrections. Some members of our fictitious open source faction will probably continue by stating that they never use software with secret sources, because it is inherently evil. They will point out the potential for manufacturers to hide spy features in closed sources, which covertly forward your computing habits to some government agency.

The other side will argue equally passionately against open source, describing its products as being of lesser quality. A well-organized development process will avoid many errors[1] from the start and well-manned quality assurance teams can catch the rest, before the software ever leaves the company. Should a vulnerability slip through and occur in the wild, the bug is reported through dedicated channels and a tested patch will be deployed via a world-wide update infrastructure. Open source software is often discredited by this group as homegrown toy-software that is written by amateurs and not fit for productive use. The many errors in such software are completely exposed, turning the public source code repositories into all-you-can-eat buffets for attackers. Because vulnerability management is not professionally organized, users are on their own to mitigate those threats.

We will revisit some of the common arguments in more detail in Section 2.

The truth unsurprisingly lies somewhere between those two extremes. People who give the matter some serious thought will weigh the pros and cons of both sides and probably answer "I don't know." It should be obvious by now that the question, whether open source or closed source is superior cannot be answered ultimately. Even the narrower question "which is more secure" has a number of facets. The surrounding issues of error introduction [29], prediction [28], detection [9], classification [41] and correction as well as fix deployment [6] and application [24] have been studied by the research community, but rarely including a specific comparison of open and closed source approaches. Error statistics are discussed comparatively [17, 19], but often in non-research contexts or even with a marketing spin. Such comparisons lack the scientific foundation that allows to draw substantiated conclusions beyond the status quo.

Contributions

We limit the broad question of relative security of open and closed source to the issue of exploitable errors in the respective software and when and by whom these errors are discovered. We *do not* consider the preceding process of how the error was introduced to the code. We equally *do not* consider the downstream issues of patch development and deployment and the time it takes to do that. We make an effort to name all our model assumptions explicitly. We think our approach of modeling error finding is novel. Different from related work in the field [1, 2], we do not base our model on error lifetimes, where probabilities are associated with the rate of errors emerging in the software. We consider the alternative of associating probabilities with the rate of errors found by an investigator. We believe this allows us to more adequately model the asymmetry between open and closed source. We compare our approach more thoroughly to related work in Section 6.

[1]In this paper, we use "bug" and "error" interchangeably. Furthermore, we are only interested in exploitable errors that constitute a security vulnerability.

We divide the people scanning a software product for errors into two groups. One group has malicious intent and watches for vulnerabilities to exploit. We call this group *attackers*. The other group has good intentions and wants to find errors to fix them and improve the software. We call them *defenders*. Our model probabilistically captures the process of those groups looking for errors. It assigns a *win probability* to each group, according to a proper definition what it means to win. Using this model, we evaluate both the open source and the closed source situation and compare the results. By *open source*, we designate a development method, where the source code is publicly readable. Thus, we use the term in a broader sense, because the original definition by Bruce Perens [32] additionally covers licensing issues such as distribution and derived work. With *closed source*, we conversely describe a development method, where source code is not available to the public, but only to certain employees of the software vendor. Such software is also labeled as "proprietary" or "binary-only".

In Section 3, we begin by presenting our assumptions and a first probabilistic model of the error-finding process. Although this introductory model is rather simple to formulate, the mathematical analysis is non-trivial. The full details of the solution can be found in Appendix A. Section 4 will extend this model to a more realistic one. Again, the detailed solution has been swapped out to Appendix B. We then use this probabilistic foundation to evaluate the core of the common arguments from Section 2, this time based on mathematical facts instead of opinions and emotions. We also demonstrate and discuss in Section 5, how the model parameters influence the outcome.

2. FACTS AND PRECONCEPTIONS

Our use of today's computer system relies on their security. Machines are exposed to a myriad of threats on the Internet and it is often debated, whether open source or closed source software is more adequate for such an environment. Both camps feature a set of archetypal arguments that are repeatedly stated by prominent advocates of the respective ideology. In this section, we briefly compile those arguments without comment and in Section 2.3 extract a common core to use for our probabilistic model in the following Section 3. More elaborate discussions have been conducted in literature [21, 31, 42].

2.1 Pro Open Source

The statements applied in favor of open source relate it to the process of peer-review used within the scientific community. Software is published for everyone to see and is validated by other members of the community, just like the papers and results of scientists undergo verification by other experts. The same review mechanisms are driving successful projects like Wikipedia. A rigorous peer-review process is used for cryptographic algorithms, where a proposed standard has to hold up for several years against continued scrutiny by experts all over the world.

The quality of peer-review improves with more people performing a review. Uncounted hobbyists all around the world are joined by paid company employees to work on open source software. In addition, students who want to learn about the inner workings of their computers turn to open source software to get insights. Open source is also often used as a teaching or research vehicle. With so many people working with the code, every hidden bug will eventually be brought to the light. Eric S. Raymond famously wrote [34]: "Given enough eyeballs, all bugs are shallow."

Prevention of back doors in open source software is held as proof for the effectiveness of the peer-review process. The convincing case of the Interbase database system is often cited [14]: A back door account to the database existed for years, but was discovered shortly after the project turned open source and published its code.

2.2 Pro Closed Source

If a bug is not discovered, there is no security problem. This is one of the arguments employed by closed source advocates. Military and government agencies have successfully relied on secret cryptographic algorithms. Keeping the source code hidden simply adds another layer of protection. Binary obfuscation techniques [23] help to further deter attackers.

The comparison to peer review in science does not hold up, because there, reviewers have a comparable level of expertise. Most users of open source software never look at the code, because they are no experts. Scientific results are only relied upon after they have undergone review. Open source software is relied upon immediately and only validated in retrospect. This defective review process provides a false sense of security to open source users. The claim that errors are found quickly by disclosing code is not justified, so it is better to hide potential vulnerabilities by keeping the source code secret.

2.3 The Argument of Diminishing Returns

Compared to closed source, an open source process helps the defenders, because there are more of them. On the other hand, an open source process helps the attackers, because it simplifies the discovery of bugs. It is interesting to see, how both sets of arguments amount to the same central point: Bugs are easier to discover in open source, thus more vulnerabilities are found. The only difference between the two camps is whether they regard this as a good thing.

Both lines of arguments bear some truth, which might explain the perpetual nature of this whole discussion. Compared to closed source, an open source approach helps both the attackers and the defenders. The real discussion should focus on the question, where the equilibrium is. The increased chance of an attacker finding an error clearly benefits all attackers. However, the larger group of defenders scales less beneficially: If one defender has a probability p to find an error, d defenders obviously do not increase this probability to dp, but rather to the combined probability $1-(1-p)^d$, showing an effect of diminishing return as the group grows.[2] The larger the group already is, the lower the benefit of adding yet another defender.

[2]**Explanation:** Probabilities only sum up for disjoint events. But for two defenders, the event of defender 1 finding a bug and the event of defender 2 finding a bug are not disjoint: The event where both find a bug is a subset of either event, so calculating the combined probability with $2p$ would account for that event twice. Here, it is conducive to look at the failure event of no defender finding a bug. For a single defender, the failure probability is $1-p$. For d defenders, all of them fail with the combined probability $(1-p)^d$. The negation of no defender finding a bug is that at least one defender finds a bug. This probability is the given $1-(1-p)^d$.

Whether the benefit for the attackers therefore outweighs the benefit of the defenders is not trivial, because this depends on various parameters. But we now have a first mathematical intuition for the problem. Modeling the error finding process should allow us to analyze the win probabilities and calculate, whether reasonable values can be reached by either group or whether one of the groups can declare victory.

3. ASSUMPTIONS AND PRECONDITIONS

Having distilled the colloquial wisdom into a concise distinction between open and closed source, we want to evaluate this mathematically. The basic distinction is summarized:

1. There are more defenders with open source compared to closed source.
2. It is easier for the attackers to find exploitable errors in open source compared to closed source.

The evaluation requires a model of the real-world error finding process. We abstract from the real world to reasonably model it probabilistically. In doing so, we make assumptions we need to formulate. For illustration purposes, we build an initial introductory model based on these assumptions. To calculate probabilities for attackers and defenders, we define, which potential outcomes are favorable for each group. The section also includes a discussion of this model's results and weaknesses.

3.1 Model Context and Assumptions

Imagine a fictitious software project existing in two variants, one developed according to the open source paradigm, the other being closed source. Both variants are independent as if they existed in different universes. Apart from the development process, they are completely identical, so any difference in their characteristics is a result of the different development approaches. Our analysis thus compares the open and closed source situation for *the same project*, it provides no insights on the relative security of *different projects*. Ranking the attacker attractiveness of different projects is an orthogonal problem. In the following, we compile the assumptions our model will rest upon.

Origins of Vulnerabilities

The software contains errors that surface as security vulnerabilities. We ignore any errors that are not exploitable, because they are of no interest for our study. Each error is introduced in the source code and the resulting vulnerability is propagated to the binary. We thus ignore vulnerabilities being introduced or masked by the compiler, because we believe those cases are extremely rare, although they do exist [43]. We similarly ignore bugs introduced by malicious [18] or vulnerable hardware [39], because to our knowledge no such exploit has been reported in the wild yet, so they currently appear without practical relevance. A vulnerability may also be introduced into a binary by an element of the runtime environment like a platform library or a byte code VM like Java. The operating system kernel is in this respect treated like a library shared by all programs. We capture such situations by choosing the faulty library or VM itself as the project we model the error finding process for. This is not exactly accurate as errors in a commonly used library can surface in any software product depending on it. We will come back to this point later in Section 5.3. For the sake of simplicity, we ignore this peculiarity for now.

Searching for Errors

Like most complex software projects, our fictitious project contains a number of exploitable errors. We assume errors to be locatable rather than ambient: You can pinpoint a specific *artifact*, be it a line of source code or an input to the executable, that triggers a vulnerability, causing the program to show behavior outside its specification. An artifact can also stand for an architectural or design weakness in the program. Other such artifacts are "clean", they do not exhibit vulnerable behavior. We assume that bug-hunters recognize a vulnerability once they have found it. We discuss in Section 5.3, how to address this assumption by modeling the expertise of the bug-hunter.

We further assume there is no way to calculate the location of errors, otherwise their removal in the development process would be trivial. Thus, errors are looked for with a trial-and-error approach. If source code is used, a tangible block of code or an individual line is picked and checked for errors. When only a binary is available, black-box testing selects some input for the program and checks the resulting behavior. The process may be supported by testbeds or analysis tools [4], but that does not change the trial-and-error nature fundamentally.

Error Distribution

Whenever an artifact is selected randomly and checked for a vulnerability, the fraction of vulnerable artifacts amongst all artifacts determines the probability that an exploitable error has been found. We apply random artifact selection, because we assume errors to be uniformly distributed across all artifacts. If errors were distributed differently, a sensible error-finding strategy would exploit knowledge about the distribution to guide the search. A uniform error distribution is a bold assumption as it is easy to imagine that errors might appear in clusters, because parts of the code might be more complex and thus more susceptible to errors [5, 10]. On the other hand, research has shown that lines of code is a good predictor for errors per source code file [28], encouraging the uniform distribution assumption at least on a large scale. Additionally, we will mitigate this assumption later in the paper. We also discuss related implications in Section 5.3.

Attackers and Defenders

Recall that two groups of people scan the software for errors with either good or malicious intentions. We call those defenders and attackers. We assume the individuals within each group to search for errors independently. There may be small clusters of attackers or defenders that cooperate internally, but our model assumes the absence of a significant large scale coordination. Defenders will always have both source and binary code at their disposal. What the attackers have available, depends on the development approach.

In the open source world, the attackers have all the same means as the defenders. Consequently, we will assign the same error finding probability to both. One could argue that the core developers of the project, who are a subset of the defenders, have an advantage, because they are intimately familiar with the project organization, its mailing lists, bug trackers and most importantly, its code. On the other hand, the attackers can fully concentrate on finding exploitable errors with no quality requirements. The defenders also deal with all the other non-security bugs, tying up a portion of their resources. To make the model feasible, we assume an independent and uniform error finding process.

In the closed source world, attackers only have access to the binary, not to source code. We therefore do not consider evil insiders, who work with malicious intent within the company and thus would have source code access. We further assume that everyone outside the company is an attacker or a passive bystander, who is of no relevance. We ignore defenders working with the binary only, like in black-box fuzz testing [27].

Influences on the Probability

The defenders receive a contribution from the outside, though. People submit bug reports, which helps to guide the error finding. Analysis tools, both static [20] and dynamic [26] point in the same direction. They help to find errors but cannot automatically and exhaustively find all of them. Similarly, attackers may have better intuition or heuristics to guide their search for vulnerabilities. We consider this as part of the error finding probabilities, but not as a fundamental piece of the model.

With open source, all tools are available to attackers and defenders, whereas closed source limits the attacker to the tools compatible with binaries. Thus, attacking an open source project is strictly easier, because all tools that work on binaries can be used by compiling the code first. In addition to that, you can use tools requiring source code. The attacker's error finding probability is therefore lower in the closed source scenario. How much lower is debatable, though. Current research has already demonstrated that exploits can be generated automatically when only a patched and an unpatched binary are available [3]. Fuzzing [27] is another potent technology attackers can leverage to find errors in binaries. During the Month of Browser Bugs [25], fuzzing impressively demonstrated its usefulness to find lots of vulnerabilities fast. Additionally, the developers may have access to in-house tools not publicly available, but this can happen in the open and closed source scenario.

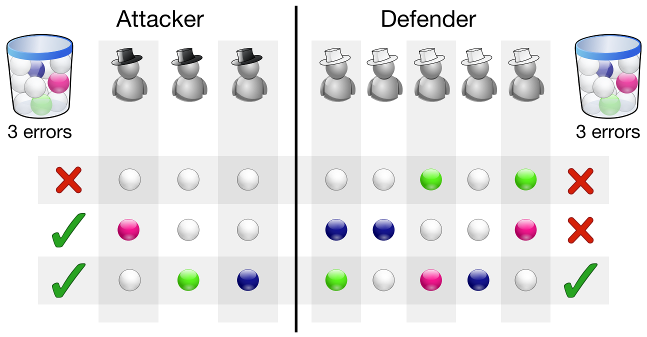
It is challenging to express all these influences with a singular error finding probability. We will therefore vary this probability in our evaluations (see Tables 1 and 2) to demonstrate the sensitivity and substantiate our conclusions.

3.2 The Static Model

To help digesting the assumptions from the preceding section we use them here by creating a simple introductory urn model[3] [16], which we call the static model. We will use the structure of this model as a foundation of the more sophisticated model in Section 4. Figure 1 provides a running illustration to support the understanding of this section.

- Each artifact (source code block, input to the binary, ...) of the software is represented by one ball in the urn. Unique colors of those balls represent the individual vulnerabilities.

[3]**Urn Model:** a thought experiment common to probability theory, where elementary events are represented by colored balls in an opaque urn. The stochastic process is represented by drawing one or multiple balls from the urn, with or without putting the balls back after they have been drawn.



The software contains three errors represented by differently colored balls. Each individual attacker and defender draws one ball from the urn. Three example outcomes are illustrated for both teams. The attackers are successful when finding any error (any colored ball). The defenders need to find every error (one ball of every color) to beat the attackers.

Figure 1: The Static Model

- Non-vulnerable artifacts are represented by white (non-colored) balls. The total number of balls determines the probability to find a ball with one specific color, i.e. one specific error.
- Attackers and defenders draw from separate urns. Each individual attacker and each individual defender is allowed to draw from the respective urn once. After drawing, the ball is immediately put back before the next individual draws. This ensures independence of the individuals and means that two individuals might independently find the same error, which we consider an effect justified in reality.

This model uses the following parameters:

p	prob. for one attacker to find a specific error
q	prob. for one defender to find a specific error
a	number of individual attackers
d	number of individual defenders
e	number of errors in the software

After executing the process and allowing each individual attacker and defender to draw once from the group's urn, we end up with each individual having a ball assigned. We can now consider, which of the potential outcomes are favorable for the attackers or the defenders.

Favorable for the Attackers

Any exploit in the wild is a failure for the security of the software. Favorable outcomes for the attackers are those, where any attacker has drawn a colored ball of any color. Hence, the probability of an outcome favorable for the attacker is:

$$p_A = 1 - (1 - ep)^a$$

Favorable for the Defenders

The defenders have only secured the software sustainably when they find all vulnerabilities. This simple verbal definition turns out difficult to formulate mathematically. Favorable outcomes for the defenders are those, where at least one ball of each color has been drawn:

$$p_D = e! \cdot \sum_{i=0}^{d-e} \binom{d}{i} q^{d-i} (1 - eq)^i \, S_{d-i,e}$$

with $S_{i,j}$ being the Stirling numbers of the second kind. Details can be found in Appendix A. This assumes there are at least as many defenders as errors ($d \geq e$).

3.3 Weaknesses

We believe this model — albeit intuitive at first glance — has major deficiencies. However, many people formulate a similar model when interviewed casually about their approach to the problem. Therefore, we think the weak points of this model can teach an important lesson.

Structural Deficiencies

The analysis of the model yields two probabilities p_A and p_D, stating the chances that an outcome of the stochastic process is favorable for the attackers or the defenders. But these two probabilities do not sum up to 1. There are outcomes that are neither favorable for the attackers nor the defenders. Does such a stalemate exist in reality? We would actually desire the favorable outcomes for the groups to be a true partition of all possible outcomes. This requires a win probability p_W and a losing probability p_L, both from the defenders point of view, that satisfy $p_W + p_L = 1$.

Semantic Deficiencies

In our construction of the favorable outcomes, we required the defenders to find all errors in one go. After a one-time starting shot, every individual group member gets one chance to look at the code and finds an error with a certain probability. After this, everything stops and we tally the result. The chances of the defenders finding all errors this way are slim. But in reality, do they actually have to find each and every error? They have to, if they want to secure the software once and for all. But software security is not an end state that can be reached. It is a process, in which the continuing existence of vulnerabilities is inevitable.

Furthermore, in reality the individual attackers and defenders do not just get one brief chance to look at the code. Instead, they scan the software repeatedly until something is found. In reality, this is a race. It is not at all important how many bugs the defenders find. The only requirement for software not to be exploited is for the defenders to find each error *before* the attackers find it. The consequence is that we need to extend the static model to include a notion of *time*, such that we can reason about probabilities for the *order* of such events.

4. MODELING THE BUG-HUNT

Before introducing a concept of time into the model, we talk about the real-world time line of a security error [11]. Afterward, we describe the dynamic model, using assumptions from the preceding static model as a foundation. We elaborate on how we enumerate the possible event orderings and which order is favorable for what group. This leads us to final winning and losing probabilities for the defenders, which satisfy $p_W + p_L = 1$.

4.1 A Bug's Life

In the good old days before the Internet became a malware zoo, the lifetime of a bug from genesis to downfall was dominated by the defenders. The software projects were buggy back then like they are today. Maybe even more so, because the developers were not as security conscious. However, bugs

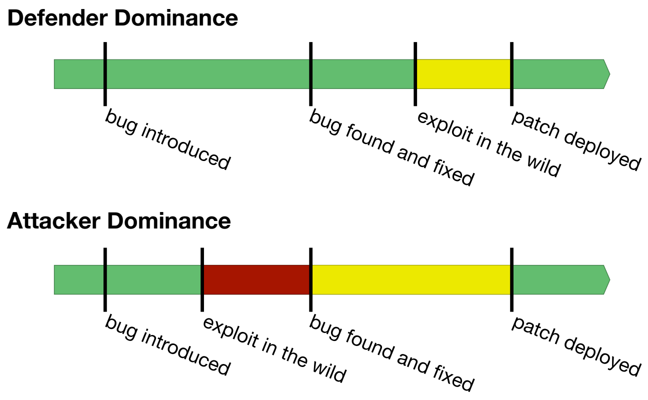


Figure 2: Time lines of a Bug's Life

were typically found by team members or indicated by outsiders through crash reports. The bug was then fixed and a patch provided, as illustrated by the upper part of Figure 2. With the availability of the patch came the attackers, who were lured by a potential exploit. This was the period when users of the software were vulnerable, because the patching culture was not as developed as it is today.

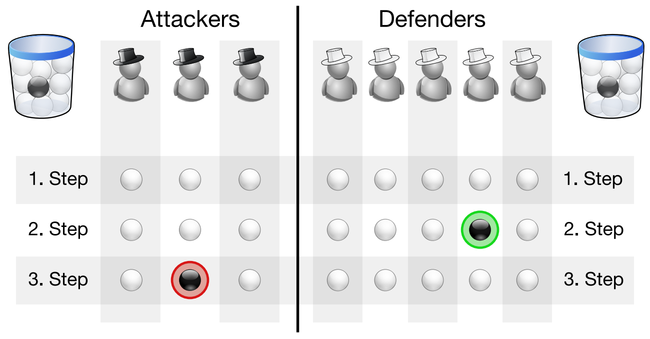
This window of exposure, where an exploit is in the wild, but users have not installed the patch has been narrowed by the online software update infrastructures we now have. Their effectiveness has been fostered by the increasing number of end users with broadband access. However, a new threat has emerged. Driven by commercial incentives, attackers became more organized and concentrate their resources on exploiting errors earlier, preferably on the day or even before the vendor has found them. In recent years, the number of zero-day exploits is increasing dramatically [11] and often, exploits are available for bugs that are not even known to the defenders yet. Such an attacker-dominated world is shown in the lower part of Figure 2. Technologies are being discussed to detect [33] and contain [6] such exploits for yet unknown vulnerabilities. The scanning activities of worms and other malware on the Internet are now so prevalent [30], the term "Internet Background Radiation" has been coined.

We think this change in the game supports our assumption that attackers and defenders operate independently. The attackers are no longer waiting for the defenders to make the first move, but search for new errors themselves. The defenders, although they get some hints from available exploits, have to debug their code on their own, because naturally, the attackers do not disclose the details of their findings to them. Today's reality also supports our intuition to model the bug-hunt as a race between attackers and defenders.

4.2 Iterative Urn Model

Our enhanced model is built on many of the same assumptions discussed earlier for the static model. A brief summary here serves as a recap:

- The object of study are two versions of the same software project with the only difference that one is open source, the other closed.
- Vulnerabilities become manifested in artifacts that can be identified in both source and binary code.



One specific error (black ball) in the software is considered. Time progresses downwards. In each step, every attacker and defender draws once from the urn. Three steps are illustrated. Whichever group finds the error (black ball) first is successful. Here, the attackers find the error in the third step, the defenders in the second. Thus, this particular outcome is favorable for the defenders.

Figure 3: The Dynamic Model

- Selecting an artifact, you can tell, whether it represents a vulnerability.
- Scanning over the artifacts is the only way to discover a vulnerability, there is no way to directly calculate which artifacts are vulnerable.
- Individual attackers and defenders independently search for errors.
- In the open case, attackers and defenders are assigned the same finding probability. In the closed case, the attackers' probability is lower.

The uniform distribution of errors is missing from the list and will be discussed later.

The race we have to model is that of finding one particular error. If the attackers find error A, the defenders must have found error A earlier to achieve a favorable outcome. Whether the defenders have found errors B and C in the meantime does not matter here, only finding or not finding error A before the attackers matters. Therefore, we now limit our discussion to one arbitrary but fixed error, ignoring all the other errors. For this one error, the question is: What are the odds for the defenders to find exactly this error before the attackers find it? We represent this situation with an urn configuration similar to the static model. Figure 3 provides a running illustration.

- Each artifact of the software is represented by one ball in the urn. The ball representing the error of concern is black, all other balls — non-vulnerable artifacts as well as vulnerable ones different from the error being considered — are white. The number of white balls in relation to the one black ball describes the probability to find this one specific error.
- Attackers and defenders draw from separate urns, because the groups independently work on different artifacts. The number of white balls in the two urns may be different, representing different error-finding probabilities.
- Individual attackers and individual defenders draw independently from their respective urn, meaning they put the drawn ball back into the urn before the next individual's turn.

We use the same parameters and symbols as described in Section 3.2 for the static model.

Time is modeled by attackers and defenders drawing from the urn *repeatedly*. The drawing steps are synchronized and can be imagined as days in the real world. In each step, each individual attacker and defender is allowed one turn at the urn and thus one chance to find the black ball. If no one has found the ball, no one has found the error, so the process continues with the next turn. We model the probability to find the ball as constant across all turns. Even if the software evolves in the meantime, we assume the error density as constant. This is supported by literature [10, 36].

As soon as anyone draws the black ball, the error is discovered, either by an attacker or by a defender. At this instance, we stop and see, which group has found the error. The defenders would now fix the error, the attackers would exploit it. Thus, any further drawing is irrelevant, because any further findings would be of no consequence. We therefore account the outcome as favorable for the group claiming the first discovery.

But how do we break the tie of both groups finding the error in the same step? Allowing this to be a neutral outcome would violate the partition requirement $p_W + p_L = 1$. Thus, we consider this outcome favorable for the attackers. They have to develop an exploit, but this can be partially automated [3]. They do not have to worry about quality testing, they can afford to deploy a half-baked exploit that is only effective on half the machines and they will still cause harm. The defenders on the other hand have to make sure their fix does not cause unwanted side-effects, they have to manage patch distribution and they rely on the end users to actually download and install the patch. We think the attackers have the upper hand here, so we account the tie as a win for the attackers.

4.3 Enumerating the Possible Outcomes

Due to the concept of time, this dynamic model has a more complex set of outcomes compared to the static model. We present the ideas behind our approach to enumerate them here and give the resulting formulas. The mathematical details can be found in Appendix B.

Firstly, imagine only *one* attacker and *one* defender, repeatedly drawing from the urn in an unbounded number of steps. The attacker's probability to find the black ball with one draw is p, the defender's probability is q. Concentrating on the attacker, the probability to find the black ball in the second step is composed of not finding it in the first step with probability $(1 - p)$ and then finding it in the second step with probability p. The compound probability to find the ball in the second step is thus $(1 - p)\, p$. Extending this to finding the ball in the m'th step means not finding it in the first $m - 1$ steps, hence the probability is $(1 - p)^{m-1} p$. This applies analogously to the defender. Because the attacker and the defender operate independently, we can calculate the probability of the attacker finding the ball in the m'th step and the defender finding the ball in the n'th step as follows:

$$p_{m,n} = (1 - p)^{m-1} p \cdot (1 - q)^{n-1} q$$

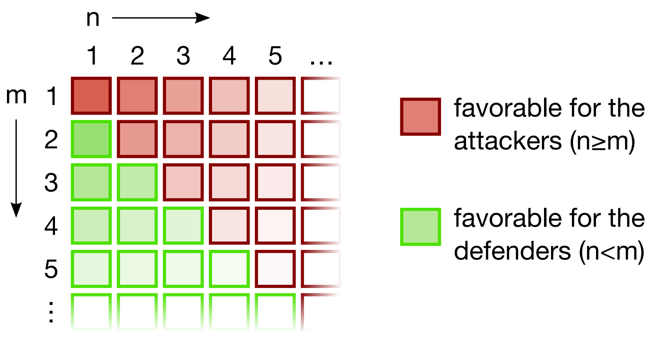

n
1 2 3 4 5 …
m 1
2
3
4
5
⋮
favorable for the attackers (n≥m)
favorable for the defenders (n<m)

Figure 4: Matrix of Possible Outcomes

As illustrated in Figure 4, we can list all possible values of $p_{m,n}$ in a matrix that is infinite in both row and column direction ($m = 1, 2, \ldots; n = 1, 2, \ldots$). Obviously, the values $p_{m,n}$ decrease with larger m or n, due to $p, q < 1$. The matrix elements therefore become smaller to the right and the bottom. Because the matrix lists all possible outcomes of the repeated drawing and these outcomes are trivially disjoint, the sum of all matrix elements $\sum p_{m,n} = 1$. A proof is given in Appendix B.

For any matrix element $p_{m,n}$ the respective outcome is favorable for the defenders, if they found the black ball before the attackers. Thus, the number n of the step, in which the defenders find the black ball must be smaller than the step number m for the attackers: $n < m$. Because we account the tie for the attackers, all other outcomes ($n \geq m$), including the diagonal are favorable for the attackers. In Figure 4, this is represented by the coloring of the matrix elements.

The win probability p_W of an outcome favorable for the defenders can now be calculated by summing up all $p_{m,n}$ with $m > n$:

$$p_W = \sum_{n=1}^{\infty} \sum_{m=n+1}^{\infty} p_{m,n} = \frac{q\,(1-p)}{q\,(1-p)+p} \tag{1}$$

Due to the construction of the matrix, $p_L = 1 - p_W$, satisfying our model requirement of $p_L + p_W = 1$ (see Appendix B for details).

One might argue now, that we modeled the process of drawing from the urn to stop, once one group has found the black ball. But here we are considering the other group to continuously draw until it too found the black ball. This appears to be a contradiction, but in fact, it is not. For example, if the defender finds the black ball in step 5, we would intuitively use the following description for the related favorable outcome: "The defender finds the ball in step five and the attacker not in steps one to five." What we actually sum up however is: "The defender finds the ball in step five and the attacker in any later step." It turns out that both formulations are equivalent. With an infinite number of draws, the attacker will eventually find the ball. Thus, "not in steps one to five" and "any step later than five" are the same.

Multiple Individuals

Generalizing from one attacker to multiple attackers is pretty straightforward. Amongst a attackers, the probability for at least one attacker to find the black ball in one draw is $\hat{p} = 1 - (1-p)^a$. Now we can just model a group of multiple attackers as one more potent attacker. The same is true for the defenders with $\hat{q} = 1 - (1-q)^d$. Replacing p and q in Equation 1 with $\hat{p}$ and $\hat{q}$ allows us to express multiple attackers and multiple defenders:

$$p_W = \frac{\hat{q}\,(1-\hat{p})}{\hat{q}\,(1-\hat{p})+\hat{p}}$$
$$\text{with } \hat{p} = 1-(1-p)^a\,,\ \hat{q} = 1-(1-q)^d \tag{2}$$

Multiple Errors

The extension to multiple errors is equally easy. If there are e errors in the software, the defenders have to find every single one of those before the attackers. Should the attackers manage to find one error earlier, security is compromised. Defenders find each individual error with probability p_W before the attackers. The probability to find all errors first is:

$$\hat{p_W} = {p_W}^{e} \tag{3}$$

Here, we again assume independent and uniformly distributed errors. But the previous results, including Equation 2 did not use assumptions on error distribution, because only one arbitrary but fixed error was considered. For the comparison of open and closed source, it is sufficient to limit the discussion to one error. If either open or closed source gets the upper hand for one error, multiple errors do not change the consequences.

5. EXPLORING THE MODEL

We now leverage our mathematical apparatus to conduct a comparison between open and closed source. We start with an analysis of Equation 2 and interpret our findings. We continue with an example software project, which we calculate for both the open and closed source cases. This section ends with possible extensions to our model that serve as starting points for future work.

5.1 Function Analysis

To get an overview on the behavior of Equation 2, we provide a graphical representation in Figure 5. The point $\hat{p} = \hat{q} = 0$ is undefined, which is intuitively clear. If no group will ever find an error, the probability of the defenders finding an error first makes no sense. Because we assume that every software has errors and that those can be found with non-zero probability, this undefined point is no problem. Other prominent values also match our intuition:

- Along the $\hat{p}$-axis, where $\hat{q} = 0$, the defenders never find any errors, so the attackers always win. Consequently $p_W = 0$ for $\hat{q} = 0$.
- On the other hand, along the $\hat{q}$-axis, where $\hat{p} = 0$, the attackers never find errors. Thus, the defenders always win ($p_W = 1$).
- Crossing the graph at $\hat{p} = 1$, p_W is always 0. How good the defenders are does not matter. If the attackers find all errors immediately, they always win.
- We broke the tie in favor of the attackers, so unsurprisingly for $\hat{p} = \hat{q} = 0.5$, p_W is lower than 0.5, because the model is not symmetric. (p_W is $\frac{1}{3}$ here.)

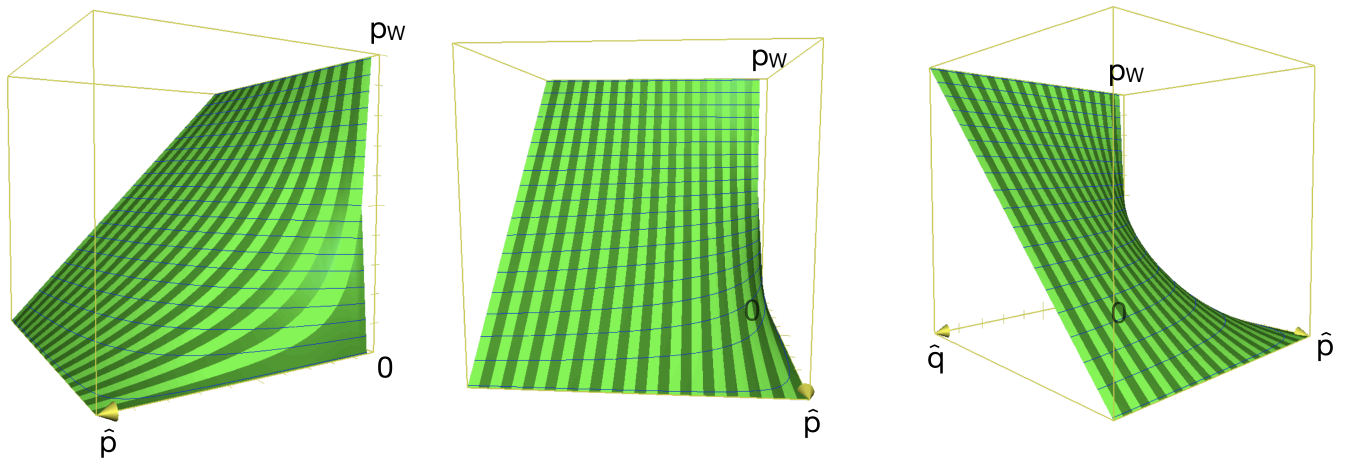


Figure 5: 3D-Plot of Equation 2 from three different angles

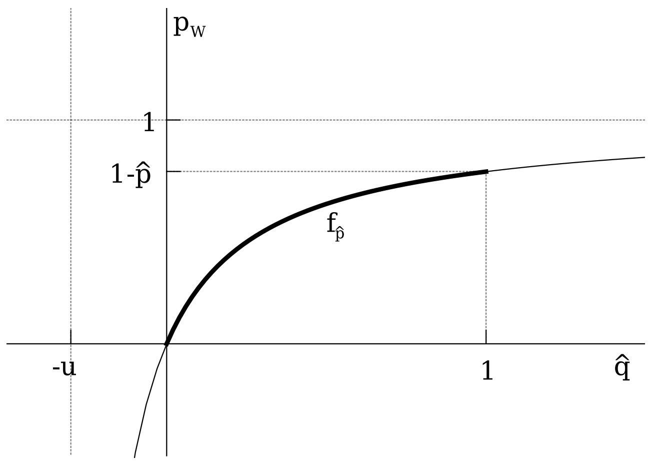


Figure 6: Defender Performance Given a Fixed p

If we reformulate p_W as a function in $\hat{q}$ with $\hat{p}$ as a parameter, we can discuss the chances of the defenders, given a fixed performance of the attackers.

$$\begin{aligned} p_W &= \frac{\hat{q}\,(1-\hat{p})}{\hat{q}\,(1-\hat{p})+\hat{p}} \\ &= 1-\frac{u}{\hat{q}+u} \\ &= f_{\hat{p}}(\hat{q}) \\ \textit{with} \qquad & u = \frac{\hat{p}}{1-\hat{p}} \end{aligned}$$

The function $f_{\hat{p}}$ is visualized in Figure 6. Two interesting results can be seen in this graph. First, $f_{\hat{p}}$ exhibits degressive growth: It climbs faster for small values of $\hat{q}$. Given that the defenders raise their $\hat{q}$ value be recruiting more developers, this result underlines the argument of diminishing returns we intuitively formulated at the beginning of the paper in Section 2.3.

Second, and maybe even more interesting, the highest win probability the defenders can achieve is lower than 1. Even though $f_{\hat{p}}$ saturates at 1 ($\lim_{\hat{q}\to\infty} f_{\hat{p}}(\hat{q}) = 1$), the highest achievable value at $\hat{q} = 1$ is:

$$p_{W,max} = f_{\hat{p}}(1) = 1-\hat{p} \tag{4}$$

Figure 5 illustrates this with the edge at $\hat{q} = 1$ starting at 1 for $\hat{p} = 0$ and ending at 0 for $\hat{p} = 1$. Interpreting this result for the bug-hunting process yields:

> The defenders cannot achieve arbitrary win probabilities, but are ultimately limited by the performance of the attackers.

No matter how many defenders work on the code, the attackers determine the envelope. This result is disturbing, because it means there will always be a window of opportunity for the attackers and there is nothing we can do about it. Within the limits and assumptions of our model, this is mathematical proof of the fact that perfect security cannot be reached outside of unrealistic corner cases (software with no errors or a world with no bad guys).

If reaching $p_{W,max}$ is the best the defenders can hope for, how far towards 1 do they have to push $\hat{q}$? Looking at the situation for $\hat{q} = \frac{1}{2}$ we get:

$$f_{\hat{p}}(\frac{1}{2}) = \frac{\frac{1}{2}\,(1-\hat{p})}{\frac{1}{2}\,(1-\hat{p})+\hat{p}} = \frac{1}{1+\hat{p}} \cdot p_{W,max}$$

Thus, if $\hat{p}$ is small, $f_{\hat{p}}(\frac{1}{2}) \approx p_{W,max}$. The defenders reach a good approximation for $p_{W,max}$ already with $\hat{q} = \frac{1}{2}$. We discuss next, how large a group is needed in a concrete scenario.

5.2 Example Software Project

We assume a fictitious software project with 1 million lines of code, which is the order of magnitude of popular open-source projects like Firefox. Existing studies report about 0.25 to 0.3 errors (including non-security bugs) in 1000 lines [7, 10], out of which 1-12% are security errors [22]. We therefore consider our project to have $e = 10^6 \times \frac{0.3}{1000} \times 0.05 = 15$ errors. This is also in line with the 0 to 0.033 vulnerabilities per 1000 lines reported by Schechter [29].

We found no conclusive studies on the number of attackers. We believe a group size of $a = 500$ is realistic for the given project size.

Table 1: Required Defenders for $\hat{p_W} = 0.6$

$\hat{p_W} = 0.6$	$q = 0.001\%$	$q = 0.002\%$	$q = 0.005\%$
$\alpha = 10$	1455	1466	1500
$\alpha = 5$	2931	2977	3126
$\alpha = 2$	7501	7815	9023
$\alpha = 1$	15630	17133	26245

Table 2: Required Defenders for $\hat{p_W} = 0.9$

$\hat{p_W} = 0.9$	$q = 0.001\%$	$q = 0.002\%$	$q = 0.005\%$
$\alpha = 10$	7360	7654	8774
$\alpha = 5$	15309	16706	24835
$\alpha = 2$	43869	62088	impossible
$\alpha = 1$	124176	impossible	impossible

The magnitude of the defender's single-step error finding probability is based on the error density of 15 security errors in 1MLOC as just established. Randomly picking a line of code will give a 0.0015% chance of hitting a vulnerable line. As discussed earlier, a lot of considerations influence this probability. Trained developers with tool support may perform better than random, so we use values of $q = 0.001\%$ to $q = 0.005\%$. To compare open and closed source, we define $\alpha = \frac{q}{p}$ to be the factor, by which the availability of source code simplifies error finding. It therefore models the advantage of the defenders, who always have source code. We vary α between 10 and 1, the latter being the open source case. Here is a summary of our model parameters so far:

e	$=$	15	number of errors
a	$=$	500	number of attackers
q	$=$	$0.001\% \ldots 0.005\%$	defender finding prob.
p	$=$	$\frac{q}{\alpha}$	attacker finding prob.
α	$=$	$10 \ldots 1$	source code benefit

We now set a target win probability $\hat{p_W}$ and then use equations 3 and 2 to calculate the minimum defender group size that reaches this probability. First, we choose a $\hat{p_W}$ of only 0.6. The chance for the defenders to find every error before the attackers is thus 60%. Even for such a disconcertingly low winning chance, the required number of defenders given in Table 1 is pretty high, especially for the open source ($\alpha = 1$) cases. Interestingly, the influence of the defender's individual finding probability q is low.

For a more realistic example, we refer to [12], which reports that for 15% of vulnerabilities found, an exploit is available before disclosure. Even though there is a gap between discovery and disclosure of a vulnerability, these 15% include the cases where the defenders lost. Thus, we use $\hat{p_W} = 0.9$ for our second example. With a nine out of ten chance, the defenders should find every error before the attackers. Table 2 shows the results. The required number of defenders grows beyond realistic values, especially for the open source scenario with a group size greater 100,000. For three cases in Table 2, a win probability of 0.9 is impossible to reach. This is a result of the upper bound determined by Equation 4. With multiple errors, this bound exponentiates to $\hat{p_{W,max}} = (1 - \hat{p})^e$. With $\alpha = 1$, $q = 0.005\%$ this upper bound is $\hat{p_{W,max}} = 69\%$. No matter how much the defenders struggle, in close to 1 out of 3 cases, the attackers will win. The only possible mitigation is to limit the attackers, from which we conclude that obscurity is the only effective deterrent available here. Additionally, a group size of 20,000 defenders already achieves $\hat{p_W} = 55\%$, whereas 40,000 defenders are needed for $\hat{p_W} = 65\%$. As seen in the previous section, the p_W grow fast at the beginning and comparatively small numbers come close to the bound already.

5.3 Potential Extensions

We designed our model to compare open source and closed source according to the process of finding errors. Along the way, we made a number of assumptions. Within the framework we provided, a number of related questions could be investigated or assumptions resolved by extending the model. We want to present some ideas here that might tip off future work.

Mitigating Assumptions

We assumed errors to adhere to an independent uniform distribution. This assumption of the static model is already partially mitigated by the dynamic model, because it regards each error in isolation and deduces, whether the attackers or the defenders find it first. As soon as finding one error simplifies finding a whole conglomerate of errors, the only requirement is that for attackers and defenders, this follow-up conglomerate is the same. If finding one error simplifies finding more errors only for one group, but not for the other, the model needs to be changed considerably to support that. The dynamic model can then no longer deal with individual errors in an isolated fashion.

We further assumed every individual attacker or defender recognizes any discovered error as such. In reality, attackers and defenders will have varying degrees of expertise and may step over an error without noticing. As a first extension to the model, errors can have individual finding-probabilities assigned that are no longer the same across all errors. This would be straightforward to incorporate into the existing model. Another possible extension could use a matrix of error-finding probabilities such that each attacker and each defender exhibits a different probability for every error. It would be tiresome to come up with values for this matrix, but it would more precisely model developer expertise. Some defenders may just not be able to find certain errors. Some attackers may ignore difficult vulnerabilities and continue searching for a lower-hanging fruit. For the model, this would change the aggregation of the individual probabilities into $\hat{p}$ and $\hat{q}$, but the calculation of p_W (see Equation 2) would still apply unaltered.

Changes to the Iterative Drawing

Remember that we broke the tie in favor of the attackers, because they do not have to worry about quality control and patch deployment but instead release their exploit immediately. The model can be extended to a larger head-start h for the attackers. Instead of $m > n$, only $m > n + h$ would then be favorable for the defenders. The basic structure of the matrix (see Figure 4) would be similar, but the triangle of the defenders would shift down. Equation 2 would require adaptation. Conversely, the defenders could be modeled with a head-start, representing the pre-release phase, where the defenders fix vulnerabilities in software not yet deployed.

Our model gives both attackers and defenders an infinite number of drawing steps. We have developed the formula for a cropped version, where the matrix is finite, because the number of steps is bounded. We did not consider the results interesting to justify inclusion in the paper.

We also briefly thought about making the steps asynchronous for attackers and defenders, e.g. for each step of the attackers, the defenders would advance two steps. We think this can be better handled by folding the two steps into one with an adequately higher success probability, so we did not pursue this modification.

Error Count

Open source projects have more defenders than closed source projects. If this increased number of code reviewers translates into a lower number of errors, and there is circumstantial evidence supporting this [35], the outcome of our model would shift towards open source. Equation 3 would need to consider separate error counts for the open and closed worlds. Further comparative studies would be necessary to answer that. Errors could also be treated according to an arrival process, with new errors being introduced to the code with a given rate and our error finding model removing errors from the system.

Peculiarities

There are some elements in the real world we dismissed in our model, because we believe them to be insignificant for our general analysis. If evidence should suggest otherwise, their inclusion in the model should be easy.

We ignored evil insiders. In a closed source environment, some attackers might actually work inside the vendor company and thus have access to the source code. Even more interesting, an evil insider can deliberately implant vulnerabilities. He later does not need to find them, because he knows of their existence. The question is, when he chooses to exploit them. For our model, this decision of the insider can be treated as a degenerated finding probability, but it may be orders of magnitude larger. The defenders have to find this implanted vulnerability like any other, but it may be particularly obfuscated. Interestingly, this situation can also arise in an open source environment.

Converting the Attackers

Our model could be used to evaluate efforts to turn attackers into defenders by providing a financial incentive. Projects like Tipping Point's Zero Day Initiative [40] pay money for reports about discovered vulnerabilities. This changes the economy of the vulnerability market and is capable of increasing the number of defenders and may even decrease the number of attackers by converting them. With our model, such efforts may analyze questions like "How many attackers do we need to convert to change the game?"

The Asymmetry of Libraries

One problem where we have not found a convincing solution within our model is the special situation of libraries. The operating system kernel can be regarded as one library common to all processes in this respect. When defenders analyze source code to check for errors, they usually stop their verification at function calls to outside libraries. They assume the library will behave according to the documentation. Of course, the library itself is also a software project, but the small team of developers working on this library defends it almost alone. The attackers however may use fuzzing techniques on the compiled binary. There, they will automatically check for errors in libraries as well. Their attacks do not stop at API boundaries but cover the entire binary code involved. This means that widely-used libraries are being pounded on by many more attackers compared to the little team of defenders.

Liability Considerations

Software security is related to the broader topic of software liability. Regarding the distinction between open and closed source software, the following observations can be made: Closed source software is often sold as a product. Even though vendors try to limit their liability in license agreements, some legal systems[4] define a set of basic liabilities that are unalienable. For some vendors of critical software, this may pose an incentive to reduce the number of errors in their products. The error discovery probabilities provided by our model could be used in a risk analysis to determine the amount of resources to invest in hardening the software. Open source software is mostly provided free of charge, constituting a gift in legal sense where liability is limited to deliberate intention or gross negligence.

Software, both open and closed source, can also be part of a service contract. A general problem with software security here is that these contracts typically cover the provisioning of a service, but security includes the absence of unwanted additional services. For open source software, the conventional wisdom of more eyeballs leading to fewer errors may cause contract partners to expect such software to have fewer bugs. We have shown that such conclusions should be handled with care. For a deeper analysis of such liability and contractual questions, our model may be integrated into larger economical or game theoretical models.

We consciously limited our view of the world to the finding of errors. Of course the discovery of a vulnerability is rather the beginning of a new line of problems. A patch must be made available and must be deployed on end-user machines. Even though timely patch distribution [6] and installation [24] is considered a systems problem, with the availability of the patch liability moves from the vendor to the end-user. However, current patch distribution mechanisms are slow compared to the ability of attackers to reverse-engineer the patch into an exploit [3].

6. RELATED WORK

We focus our discussion of related papers to vulnerability analysis using probabilistic or stochastic approaches. Related work in a broader context — from debugging tools to patch deployment — has been mentioned throughout the paper already.

In [29], Ozment and Schechter provide a detailed and elaborate analysis of the introduction and reporting of vulnerabilities in the OpenBSD operating system. A period of 7.5 years with 15 versions is considered. Supporting the numbers we used in our evaluation, the authors state that vulnerability densities are orders of magnitude lower than overall error densities, which include bugs that are not vul-

[4]Being German authors, we are more familiar with the German legal system. Arguments given here should be understood in this light.

nerabilities. Overall, the vulnerability densities range from 0 to 0.033 per thousand lines of code. Furthermore, Ozment and Schechter found evidence that the number of vulnerabilities decreases with the lifetime of the project. However, that decline is not very pronounced.

Rescorla [36] could not find evidence for a downward trend in vulnerabilities over the lifetime of software. He conducts a cost-benefit analysis of vulnerability disclosure for a large base of software and with two different exponential models. His data does not contradict the possibility of a constant vulnerability rate, which is an assumption we use in our model.

Several papers [1, 2] describe the emergence of errors with software reliability growth models. Errors appear randomly, with the rate modeled after the mean time between failure. Anderson [2] uses an exponentially distributed error lifetime and derives an approximated polynomial error distribution, the so called thermodynamic model. In this model, attackers and defenders appear as passive observers of the software that randomly shows errors. Consequently, Anderson concludes that making it easier or harder to find attacks helps attackers and defenders equally. He goes on to discuss other effects that break the symmetry, focusing primarily on trusted computing.

Alhazmi and Malaiya employ the same error lifetime paradigm in [1]. They analyze various models for the error rate by fitting them to data from real systems. Interestingly, Anderson's thermodynamic model was the least successful in matching the empirical data, which emphasizes the difficulty to model such a complex process. We think the approaches based on error lifetime are not well-suited for comparing open and closed source systems. Describing the emerging errors as a probabilistic process does not offer a concise way to model the asymmetric situation of attackers and defenders in open- and closed-source scenarios. Our model thus employs the complementary view of associating probabilities to the attackers and defenders, who actively search for errors. This interpretation allows us to model the bug-finding race between the two groups with their differing conditions appropriately.

Vulnerability discovery and security implications in general have been analyzed economically [37, 15]. Such models can help ranking potential attacker targets, as the return gained from exploiting a particular software depends on the potential damage and the associated cost. But these discussions are orthogonal to our results, because we compare open and closed situations for the same software. Our findings can help deciding whether a codebase should be kept secure or opened to the public. We cannot say which of two different projects — independent of open or closed source — is more likely to be exploited.

7. CONCLUSION

The central result from our analysis of the dynamic model is that obscurity is an effective deterrent and the larger number of defenders in the open source case does not generally suffice to counteract this. The power of the attackers ultimately limits the power of the defenders, so they cannot achieve arbitrary winning chances. The significance even of a small limit is exponentiated by the fact that defenders have to find each error before the attackers. These facts appear to speak against open source, but in reality, one should never rely on either obscurity or openness alone, but consider the bigger picture [21]. Therefore, we want to take a closer look at the open source environment and its numerous benefits before we summarize the paper.

7.1 A Case for Open Source

First of all, this paper solely considers the discovery of a vulnerability, not what follows after that. The entire discussion about vulnerability disclosure policies and their benefits and downsides is therefore out-of-scope for this work. If a closed source vendor learns about a security issue and chooses not to fix it immediately, but collect more errors for a cumulative patch, this changes the game in favor of open source. Publishing information about discovered vulnerabilities creates an incentive for the vendor to fix bugs early [38]. Hiding such information is dangerous, because the vulnerability has already been found and the defenders are needlessly testing their luck.

Other benefits of open source stem from the distribution of knowledge it entails. It encourages curious users who want to learn more about the foundations of their system. If they want to experiment, they have the freedom to change things as they please. For that reason, open source software often serves as valuable research vehicles. In trust-critical environments, open source enables stakeholders to perform their own security audits without the need to trust outside third parties. This is especially interesting for countries that distrust binary software from potential enemies.

One fundamental assumption in our paper is that *the same project* exists in an open and a closed source version and we compare the two. In reality, this does not exist. The open source development methodology does not share the deadline-pressure pushing commercial vendors to release software that is not fully tested and reviewed. And because the pre-release and post-release error density can differ by an order of magnitude [10], a testing phase dictated by technical instead of market considerations is certainly helpful. Studies have shown that closed source vendors' patching efforts slow down before major product releases [13].

7.2 Summary

In this paper, we contribute to answering the question: "Is open source more secure?" We limit this very broad question to the process of discovering vulnerabilities in software and whether the attackers or the defenders find errors first. The contribution of our work is a mathematical model of this asymmetric race. We think the model may also be applicable in the broader context of published vs. secret proofs or cryptographic algorithms, but we have not thought this through yet.

Surprisingly, the colloquial argument that open source is superior given a large number of bug-hunters is not justified by our model. First, even with an arbitrarily large number of defenders, the attackers cannot be dominated unboundedly. The power of the defenders has an upper bound ultimately dictated by the attackers. Second, with realistic numbers we show that the defenders do not require excessively large groups to approach this bound.

We conclude that open source is not intrinsically more secure. Obscurity can be an effective deterrent for attackers. However, open source has profound advantages beyond mere vulnerability chances, ranging from research to information freedom. Thus, the authors do not consider this paper an argument to abandon the open source idea, but rather as a contribution to understand its limits.

Acknowledgments

We thank our colleague Dr. Jürgen Brunner for pointing us to the Stirling numbers to solve our combinatorial problem in the static model. We further thank Prof. Dr. Andreas Westfeld for his comments on software liability. We thank our coworkers Björn Döbel and Stefan Kalkowski who read the paper and provided valuable feedback. We also thank our shepherd Nicolas Christin and the anonymous reviewers for all their comments and suggestions which greatly helped improving the paper.

8. REFERENCES

[1] Alhazmi, O. H., and Malaiya, Y. K. Modeling the Vulnerability Discovery Process. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 129–138.

[2] Anderson, R. Security in Open versus Closed Systems – The Dance of Boltzmann, Coase and Moore. In *Open Source Software: Economics, Law and Policy* (2002), pp. 127–142.

[3] Brumley, D., Poosankam, P., Song, D., and Zheng, J. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 143–157.

[4] Cadar, C., Dunbar, D., and Engler, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08: Proceedings of the eighth USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 209–224.

[5] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An Empirical Study of Operating Systems Errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM, pp. 73–88.

[6] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: End-to-End Containment of Internet Worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 133–147.

[7] Coverity. Open Source Report. `http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2008.pdf`, 2008.

[8] Dossey, J. A., Otto, A. D., Spence, L. E., and Eynden, C. V. *Discrete Mathematics*. Pearson Education Inc., 2006.

[9] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM, pp. 57–72.

[10] Fenton, N. E., and Ohlsson, N. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering 26*, 8 (2000), 797–814.

[11] Frei, S., May, M., Fiedler, U., and Plattner, B. Large-Scale Vulnerability Analysis. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense* (New York, NY, USA, 2006), ACM, pp. 131–138.

[12] Frei, S., Schatzmann, D., Plattner, B., and Trammell, B. Modelling the Security Ecosystem – The Dynamics of (In)Security. In *Workshop on the Economics of Information Security (WEIS 09)* (2009).

[13] Frei, S., Tellenbach, B., and Plattner, B. 0-Day Patch – Exposing Vendors (In)security Performance. `http://www.blackhat.com`, 2008.

[14] Havrilla, J. S. Cert advisory ca-2001-01: Interbase server contains compiled-in back door account. `http://www.cert.org/advisories/CA-2001-01.html`, January 2001.

[15] Herley, C. So Long, And No Thanks for the Externalities: The Rational Rejection of Security Advice by Users. In *Proceedings of the New Security Paradigms Workshop (NSPW 09)* (New York, NY, USA, 2009), ACM.

[16] Johnson, N. L., and Kotz, S. *Urn Models and Their Application*. John Wiley & Sons, New York, USA, 1977.

[17] Jones, J. Browser Vulnerability Analysis of Internet Explorer and Firefox. `http://blogs.technet.com/security/archive/2007/11/30/download-internet-explorer-and-firefox-vulnerability-analysis.aspx`, 2007.

[18] King, S. T., Tucek, J., Cozzie, A., Grier, C., Jiang, W., and Zhou, Y. Designing and Implementing Malicious Hardware. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–8.

[19] Krebs, B. Internet Explorer Unsafe for 284 Days in 2006. `http://blog.washingtonpost.com/securityfix/2007/01/internet_explorer_unsafe_for_2.html`, January 2007.

[20] Kremenek, T., Twohey, P., Back, G., Ng, A., and Engler, D. From Uncertainty to Belief: Inferring the Specification Within. In *OSDI '06: Proceedings of the seventh USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 161–176.

[21] Levy, E. Wide open source. `http://www.securityfocus.com/news/19`, April 2000.

[22] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., and Zhai, C. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (New York, NY, USA, 2006), ACM, pp. 25–33.

[23] Linn, C., and Debray, S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In

CCS '03: Proceedings of the 10th ACM conference on Computer and communications security (New York, NY, USA, 2003), ACM, pp. 290–299.

[24] Makris, K., and Ryu, K. D. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 327–340.

[25] Moore, H. D. Month of Browser Bugs. `http://browserfun.blogspot.com/`.

[26] Nethercote, N., and Seward, J. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), ACM, pp. 89–100.

[27] Oehlert, P. Violating Assumptions with Fuzzing. *IEEE Security and Privacy 3*, 2 (2005), 58–62.

[28] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. Where the Bugs Are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2004), ACM, pp. 86–96.

[29] Ozment, A., and Schechter, S. E. Milk or Wine: Does Software Security Improve with Age? In *Proceedings of the 15th USENIX Security Symposium (USENIX-SS 06)* (Berkeley, CA, USA, 2006), USENIX Association.

[30] Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L. Characteristics of Internet Background Radiation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2004), ACM, pp. 27–40.

[31] Payne, C. On the security of open source software. *Information Systems Journal 12* (2002), 61–78.

[32] Perens, B. The Open Source Definition. `http://www.opensource.org/docs/osd`.

[33] Provos, N. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association.

[34] Raymond, E. S. The cathedral and the bazaar. `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`, 2000.

[35] Reasoning, LLC. How Open Source and Commercial Software Compare: MySQL 4.0.16. `http://www.reasoning.com/pdf/MySQL_White_Paper.pdf`, 2006.

[36] Rescorla, E. Is Finding Security Holes a Good Idea? `http://www.rtfm.com/bugrate.html`, 2005.

[37] Schechter, S. E. Toward Econometric Models of the Security Risk from Remote Attacks. *IEEE security & privacy* (2005), 40–44.

[38] Schneier, B. The Nonsecurity of Secrecy. *Communications of the ACM 47*, 10 (2004), 120.

[39] Shamir, A. Research Announcement: Microprocessor Bugs can be Security Disasters. `http://cryptome.info/bug-attack.htm`, November 2007.

[40] TippingPoint Technologies, Inc. Zero Day Initiative. `http://www.zerodayinitiative.com/`.

[41] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Triage: Diagnosing Production Run Failures at the User's Site. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 131–144.

[42] Wheeler, D. A. Secure Programming for Linux and Unix HOWTO: Is Open Source Good for Security? `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/open-source-security.html`, March 2003.

[43] Xfocus Security. BCB compiler incorrect deal sizeof operator vulnerability. `http://www.xfocus.org/advisories/200603/8.html`, March 2006.

APPENDIX

A. DETAILED SOLUTION FOR THE STATIC MODEL

We seek the probability $P(n,k,p)$ of n individuals finding k different errors. Each individual finds one error with probability p, satisfying $n \geq k$ and $kp \leq 1$. Hence, with the remaining probability $1-kp$ the individual finds no error.

1. The number of cases of n individuals finding k errors $F_1 \dots F_k$ in this order is equal to the number of cases to assign n different balls in k different urns, with no urn staying empty. This number is given exactly by the Stirling numbers of the second kind $S_{n,k}$ [8]:

$$S_{n,k} = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n, \quad n \geq k \geq 0$$

Errors can be found in any order. Thus, this number increases to $k! \cdot S_{n,k}$. Assuming that *each* individual finds an error (with probability p), the associated probability is:

$$k! \cdot S_{n,k} \cdot p^n$$

2. Now, $1, 2, \dots, n-k$ individuals may not find an error (with more than $n-k$ individuals not finding an error, not all k errors can be found). There are $\binom{n}{i}$ possibilities for i out of n individuals to find no error (with probability $1-kp$). In analogy to 1, there are $k! \cdot S_{n-i,k}$ cases for the remaining individuals to find all k errors. Hence, the respective probability equals to:

$$\binom{n}{i} \cdot k! \cdot S_{n-i,k} \cdot p^{n-i}\,(1-kp)^i$$

3. Finally:

$$P(n,k,p) = k! \cdot \sum_{i=0}^{n-k}\binom{n}{i}p^{n-i}\,(1-kp)^i \cdot S_{n-i,k}$$

B. DETAILED SOLUTION FOR THE DYNAMIC MODEL

The matrix of possible outcomes is calculated as follows:

$$p_{m,n} = (1-p)^{m-1} \cdot (1-q)^{n-1} \cdot pq$$

The matrix thus has the following structure:

$$\begin{bmatrix} pq & p\bar{q}q & p\bar{q}^2q & p\bar{q}^3q & \cdots \\ \bar{p}pq & \bar{p}p\bar{q}q & \bar{p}p\bar{q}^2q & \bar{p}p\bar{q}^3q & \cdots \\ \bar{p}^2pq & \bar{p}^2p\bar{q}q & \bar{p}^2p\bar{q}^2q & \bar{p}^2p\bar{q}^3q & \cdots \\ \bar{p}^3pq & \bar{p}^3p\bar{q}q & \bar{p}^3p\bar{q}^2q & \bar{p}^3p\bar{q}^3q & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$\text{with } \bar{p} = 1-p,\ \bar{q} = 1-q$$

1. Now we sum up the outcomes favorable for the defenders column-wise:

$$\begin{aligned} p_W &= \sum_{n=1}^{\infty}\sum_{m=n+1}^{\infty} p_{m,n} \\ &= pq\cdot\Big[\bar{p}\left(1+\bar{p}+\bar{p}^2+\ldots\right)+ \\ &\quad \bar{p}^2\bar{q}\left(1+\bar{p}+\bar{p}^2+\ldots\right)+ \\ &\quad \bar{p}^3\bar{q}^2\underbrace{\left(1+\bar{p}+\bar{p}^2+\ldots\right)}_{\frac{1}{1-\bar{p}}=\frac{1}{p}}+\ldots\Big] \\ &= pq\cdot\bar{p}\cdot\frac{1}{p}\cdot\Big[1+\bar{p}\bar{q}+\bar{p}^2\bar{q}^2+\ldots\Big] \\ &= q\bar{p}\cdot\frac{1}{1-\bar{p}\bar{q}} \\ &= \frac{q\,(1-p)}{1-(1-p-q+pq)} \\ &= \frac{q\,(1-p)}{1+p+q-pq} \\ &= \frac{q\,(1-p)}{q\,(1-p)+p} \end{aligned}$$

2. Further, we show that the sum of all $p_{m,n}$ is 1:

$$\begin{aligned} \sum_{m,n} p_{m,n} &= \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} pq\,(1-p)^{m-1}\,(1-q)^{n-1} \\ &= pq\cdot\sum_{m=1}^{\infty}\left[(1-p)^{m-1}\cdot\sum_{n=1}^{\infty}(1-q)^{n-1}\right] \\ &= pq\cdot\sum_{m=1}^{\infty}(1-p)^{m-1}\cdot\sum_{n=1}^{\infty}(1-q)^{n-1} \\ &= pq\cdot\sum_{m=0}^{\infty}(1-p)^{m}\cdot\sum_{n=0}^{\infty}(1-q)^{n} \\ &= pq\cdot\frac{1}{1-(1-p)}\cdot\frac{1}{1-(1-q)} \\ &= pq\cdot\frac{1}{p}\cdot\frac{1}{q} \\ &= 1 \end{aligned}$$

Hence, due to the structure of the matrix, $p_L = 1-p_W$.