Protection Poker: The New Software Security "Game"

Without infinite resources, software development teams must prioritize security fortification efforts to prevent the most damaging attacks. The Protection Poker "game" is a collaborative means for guiding this prioritization and has the potential to improve software security practices and team software security knowledge.

Laurie Williams and Andrew Meneely
North Carolina State University

Grant Shipley
Red Hat

Playing cards in hand, the software development team members stare silently at their cards. Players glance at each other while pensively considering their options. Grant, the development manager, announces, "Everybody ready?" and each member lays down a card. At once, the silence erupts into a team-wide conversation of opinions, perspective, and debate.

No, this isn't your secret lunchtime poker game in the broom closet. Nor is this a naïve "team-building" activity from human resources. The team is playing Protection Poker,[1] a new software security "game."

Protection Poker is an informal game for security risk estimation that leads to proactive security fortification during development and prioritizes security-related validation and verification (V&V). Protection Poker provides structure for collaborative misuse-case[2] development and threat modeling[3,4] that plays off the participants' diversity of knowledge and perspective. The entire extended development team gets involved—software developers, testers, product managers or business owners, project managers, usability engineers, security engineers, software security experts, and others. Protection Poker is based on a collaborative effort estimation practice, Planning Poker,[5] which many agile software development teams use. (The "rules" of Planning Poker don't at all resemble actual poker's rules, except that each participant hides his or her cards from the other participants until a designated time. Collocated teams often use special cards to do their estimation that contain only selected values. See the sidebar "Protection Poker's Historical Roots" for more information).

Here, we explain Protection Poker and discuss a case study Red Hat IT software engineers conducted. The Red Hat IT team utilized the Scrum[6] agile software development methodology and "played" Protection Poker during its biweekly iteration planning meetings over a four-month period.

Protection Poker

Protection Poker is a simple but effective software security game. Its tangible output is a list of each requirement's relative security risk. The team can use this relative risk to determine the type and intensity of design and V&V effort the development team must include in the iteration for each requirement. The team can then use this list to help prioritize security engineering resources toward software areas with the highest risk of attack based on factors such as how easy the new functionality is to attack and the value of the data accessed through the functionality. Consequently, the team properly estimates the necessary effort to implement the requirement securely, so it can proactively plan which resources are needed for secure implementation. This prioritization and increased knowledge should lead a team toward developing more secure software. Protection Poker works best for teams that use an iterative development process with relatively short iterations, as agile software development teams often do.

How Do You Play?

Teams play during an iteration planning meeting, in which the focus is on the specific requirements the

team is likely to choose to implement during the upcoming iteration. The game involves the following process for each requirement. First, the product manager or business owner explains the requirement to the team. Conversations involve clarifying the expectations for the requirement until the team has no more questions.

Subsequently, the team discusses the new requirements' security implications on the evolving system. Will implementing it make the system more or less vulnerable to attack? Or, perhaps, is the new functionality so hidden from a potential attacker that the system's security posture remains unchanged? Informal discussions about misuse cases and threat models ensue. The moderator for the iteration planning meeting can instigate discussion by asking leading questions such as, "Who would want to attack the system?"; "What could an attacker do if they got a hold of the data stores this new requirement accesses—and stole, deleted, or corrupted the data?"; and "What damage could an insider do through this functionality, particularly if he or she could bypass the user interface?" These discussions might reveal, for example, that the role-based access to some functionality needs to be more restrictive or that logging would provide valuable information in light of an insider attack.

When the conversation about security implications quiets down, the team moves toward the next phase of the game—voting on the security risk components. We traditionally compute risk[4,7] as follows:

Risk = (probability of loss) * (impact of loss).

Protection Poker uses a variation of this computation to determine security risk:

Security risk = (*ease* of attack) * (the *value* of the asset that could be exploited with a successful attack).

This computation is based on the hypothesis that attackers are more likely to succeed in attacking features that are of high value and easier to attack. The first part of the equation relates to how hard or easy the new, enhanced, or corrected functionality makes it to attack the system. The easier an attack could be, the higher the probability that one will occur and the greater the risk. Additionally, an attack's probability increases if the adversary finds the assets accessible via the new functionality to be attractive or valuable and is thus more willing to work hard at devising an attack. The loss or compromise of an asset that's attractive to an adversary is also likely to have a larger impact on the organization.

Inspired by Planning Poker (see the sidebar), Protection Poker uses the relative measures of *ease points* and *value points* in its security risk computation (for example, one requirement is five times easier to attack than another). Team members vote on their estimate of relative measures for ease of attack and asset value. The team is constrained to nine possible values—for instance, 1, 2, 3, 5, 8, 13, 20, 40, and 100 (which Planning Poker uses)—for ease points and value points. The game uses these particular values because humans are more accurate at estimating small things; hence more possible small values exist than large ones.[5] Additionally, team members can do their estimations more quickly with a limited set of possible values. For example, why argue over whether a requirement is 40 or 46 times easier to attack than another? At that point, we can only really know that the requirement is "a lot easier" to attack.

Ease of Attack

Let's say the team decides to vote on ease points first. Earlier security discussions will likely have covered the ease of attack in general. A short, more focused discussion on ease is likely to occur. Team members

Protection Poker's Historical Roots

Protection Poker and Planning Poker are Wideband Delphi techniques.[1] (Planning Poker's creator likely chose its name for the catchy alliteration, whereas the term Wideband Delphi might have seemed less accessible to agile teams.) Wideband Delphi is based on the Delphi practice,[2] developed at the RAND Corporation in the late 1940s for the purpose of making forecasts. With the Delphi practice, participants make estimates individually and anonymously in a preliminary round. They collect, tabulate, and return the first-round results to each participant for a second round, during which they must again make a new forecast regarding the same issue. This time, each participant knows what the other participants forecasted in the first round, but doesn't know the other participants' rationale behind those forecasts. The second round typically results in a narrowing of the group's range in forecasts, pointing to some reasonable middle ground regarding the issue of concern. The original Delphi technique avoided group discussion[3] to enable candid and anonymous input.

Barry Boehm created the Wideband Delphi technique[1] as a variant of the Delphi technique where group discussion occurs between rounds in which participants explain why they've chosen their values. Wideband Delphi is useful for coming to some conclusion regarding an issue when the only information available is based more on experience than empirical data.[3]

References

1. B.W. Boehm, *Software Eng. Economics*, Prentice-Hall, 1981.
2. U.G. Gupta and R.E. Clarke, "Theory and Applications of the Delphi Technique: A Bibliography (1975–1994)," *Technological Forecasting and Social Change*, vol. 53, no. 2, 1996, pp. 185–211.
3. B. Boehm, C. Abts, and S. Chulani, "Software Development Cost Estimation Approaches—A Survey," *Annals of Software Eng.*, vol. 10, nos. 1–4, 2000, pp. 177–205.

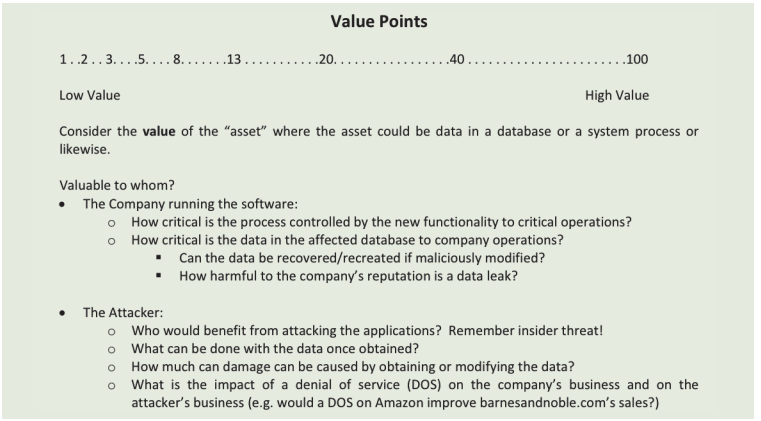

Figure 1. Sample "cheat sheet" of security issues. Protection Poker participants can be reminded of the scoring system along with common threats by looking at a cheat sheet. (Available at http://collaboration.csc.ncsu.edu/laurie/Security/ProtectionPoker/MemoryJogger.pdf.)

can aid such discussion using a "cheat sheet" of common security issues. Figure 1 shows a sample cheat sheet. Team members often make statements such as,

- "<new requirement> increases the attack surface as much as does <other requirement>" (the attack surface is the union of code, interfaces, services, and protocols available to all users, especially those that unauthenticated or remote users can access[3]); or
- "the option to execute this function only appears on the user interface for the admin due to our role-based access control."

Additionally, the discussions might lead the team to revise a requirement on the spot so as to reduce risk. For example, the team could decide to log the user id for all database updates and deletions. The team moderator would promptly update the documentation for the requirement to record this new logging task, which reduces the ease-point estimate.

When the team has exhausted comments and questions on ease, a vote takes place. Each team member chooses a card from a specialized deck containing the values 1, 2, 3, 5, 8, 13, 20, 40, and 100. When a team member chooses a value, he or she is thinking about ease points relative to the previously estimated requirements' ease points (that is, "This requirement is about as easy to attack as <some other requirement> and we gave that one a value of 5," or "This is 20 times easier to attack than <other requirement>"). The team members reveal their estimates simultaneously.

Differences of opinion often reveal misunderstandings and new perspectives and perceptions. First, the team members with the lowest and highest estimate explain them to the group, after which an open discussion follows until the group is ready to revote on their estimates. More estimation rounds occur until the team can agree on ease points for the requirement. Most often, two or three Protection Poker rounds are necessary on a particular requirement before the team reaches consensus.

Value of Assets

Next, a Protection Poker round takes place to estimate a requirement's value points based on the assets the new requirement protects or uses. Typically, these assets include data stored in database tables or system processes that the new functionality controls. The value of a particular asset, such as a database table, doesn't change based on the various requirements that use that asset. So, the team stores and reuses the value-point estimate for assets.

The value-point estimation discussion begins with everyone in the room enumerating the assets they feel the requirement "touches" or creates. The team discusses and votes on a value-point estimate for any new asset and for any existing asset that doesn't yet have a value-point estimate. For example, the team might decide that the password table is 100 times more valuable to an attacker than the table containing statistics about baseball players. The password table would get a value of 100 and the baseball player table a value of 1. When value-point estimates exist for all enumerated assets, the moderator sums them as the value-point estimate for the requirement.

A variation of the adage "When everything is high priority, then nothing is high priority" is "When everything is very valuable, then nothing is very valuable." Consequently, the team is best served by actually differentiating the value of the assets the software system uses.

Calculating Risk

A Protection Poker discussion is driven by the diversity of participant opinions about the ease of attack and the value of the protected asset for a requirement. Gary McGraw calls this type of discussion *ambiguity analysis*,[8] which is the subprocess capturing the creative activity required to discover new risks when one or more participants share their divergent understandings of a system. Disagreements and misunderstandings often indicate security risk.[8] Participants shouldn't implicitly or explicitly be coerced into agreeing with the estimate from those considered the most important or most respected. As such, a culture that values a diversity of opinions is necessary for Protection Poker to be an effective technique.

Teams should calculate ease- and value-point values at the project's start, such that an implementation of a requirement that would be very difficult to attack—for example, because it doesn't increase

the attack surface—receives a value of 1, whereas an implementation of a requirement that would be easy to attack receives a 40 or a 100. The team estimates all other requirements relative to these end points. The calibration can change throughout multiple iterations.

We compute a requirement's risk profile by multiplying ease points by value points, as Table 1 shows. The development team can use this profile to prioritize software security efforts, such as a more formal development of misuse cases or threat models; security inspection; security redesign; the use of static analysis tools; and security testing. The team factors a requirement's relative risk and the resulting actions necessary to implement it into the effort estimate for implementing the requirement. It can also factor the necessary software security effort into the overall effort estimate such that resources are allocated for and realistic completion times are committed to implementing a secure requirement. Furthermore, the final risk results can elicit more security discussion because they're sometimes surprising. In the example Table 1 shows, the most valuable asset didn't pose the highest risk when used in requirement 1. Rather, requirement 7 had the highest risk because it's easy to attack, and it used an asset with a high value.

As we've mentioned, through the structured Protection Poker conversation, the extended team discusses the ease and value points for each requirement in turn. The team shares the proposed requirements' business and technical details, such as the assets the requirement will use (for example, sensitive personal information in a database) and discusses the technical risks that jeopardize business details. For instance, a requirement that necessitates a screen with 15 user input fields that are used in dynamic SQL queries is inherently more risky than one that involves only behind-the-scenes batch processing. Similarly, high-usage requirements impose a denial-of-service (DoS) risk. The structured discussion about security issues that occurs during Protection Poker should greatly improve the team members' knowledge and awareness of what is required to build security into the product.

Case Study: Red Hat IT

We conducted a case study of Protection Poker with a Red Hat IT maintenance team in Raleigh, North Carolina. The team supports 37 Web-based products, all of which are written in Java (J2EE), Python, Perl, and PL/SQL-stored procedures. The applications are large, often more than 500,000 lines of code.

The team comprised 11 software engineers: seven developers, three testers, and one manager. All but one member lived in Raleigh. However, the engineers worked from home approximately half the time. An additional four business analysts provided input and prioritization to the team. The team didn't have a dedicated security expert nor did any security experts participate in the Protection Poker sessions. We observed that the team members had a wide range of software security knowledge; some were quite knowledgeable, whereas others were relatively new to software security practices. During Protection Poker sessions, all team members participated in the conversation—some asking questions, some sharing their software security expertise, all becoming incrementally better at thinking like an attacker with each Protection Poker session. For example, the team initially trusted the users if Red Hat employees were to run the functionality on an intranet, but eventually learned to consider insider attack.

Table 1. Prioritizing risk with Protection Poker.

Requirement	Ease points	Value points	Security risk	Rank
1	1	1,000	1,000	2
2	5	1	5	6
3	5	1	5	6
4	20	5	100	4
5	13	13	169	3
6	1	40	40	5
7	40	60	2,400	1

In general, the team follows the Scrum[6] software development process. In a typical iteration-planning meeting, team members discuss 10 requirements. Because the Red Hat team is a maintenance team, these requirements are problem reports Red Hat employees or external users have submitted (the latter via a help desk). These problem reports would equate to a functional requirement in a team developing a new product or a new release of an existing product.

We introduced Protection Poker to the Red Hat IT team via a 90-minute tutorial led by Laurie Williams. After an enthusiastic response to Protection Poker based on the tutorial, we planned to conduct Protection Poker sessions with the 11-person team during their iteration planning meetings held every two weeks. We studied the Red Hat IT team's use of Protection Poker for three months. During this time, the two of us from North Carolina State University (Laurie Williams and Andrew Meneely) participated in five Protection Poker sessions at Red Hat that the third author, Grant Shipley, led. We periodically interjected procedural suggestions and potential security considerations. We also recorded data and observations about the sessions. The team completed a short survey administered via surveymonkey.com after completing the initial tutorial and participating in three Protection Poker sessions.

As with all case study research, the results of one industrial team might not be representative of what we'd observe in another. The team we studied is re-

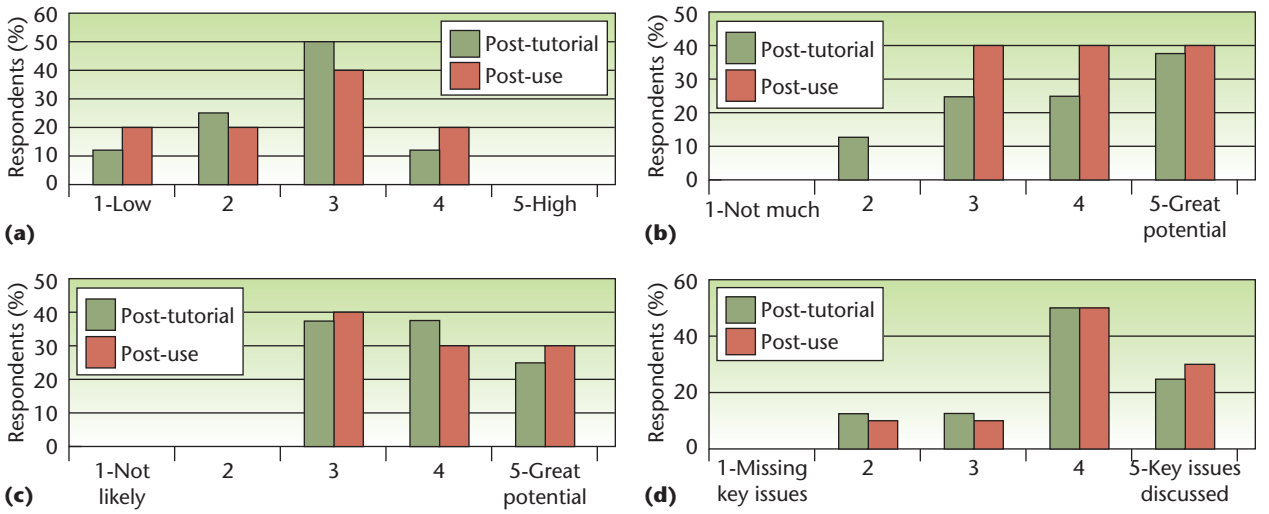

Figure 2. Histograms of Red Hat IT survey results. We conducted surveys both post-tutorial (before using Protection Poker) and post-use (after using Protection Poker). We asked for feedback on the following statements: (a) rate your software security knowledge; (b) Protection Poker will help you learn about security; (c) Protection Poker will help spread security knowledge throughout your team; and (d) Protection Poker focuses discussion on what you feel are the true security risks.

sponsible for maintaining Red Hat IT products, not new code development. Our study doesn't report on changes in field quality related to the security of the products the Red Hat team supports. Due to the latency of maintenance releases and customer-discovered vulnerabilities, the long-term security impact of Protection Poker's use might not be statistically evident for more than a year.

Results

The Protection Poker sessions yielded the following actionable outcomes:

- The team revised two requirements for added security fortification. One change involved additional logging so that the system administrators could obtain an audit trail of access. Team members revised the other requirement to explicitly state the need to prevent cross-site scripting vulnerabilities.
- During the meeting, an engineer launched a successful cross-site scripting attack to the application under discussion after the team completed the first group discussion about cross-site scripting. The same engineer then asked for an education session on preventing cross-site scripting vulnerabilities.
- A discussion ensued about the need for a governance process to prioritize fortifying identified security vulnerabilities over fixing noncritical reliability defects.
- A business analyst suggested that all new requirements testing involve additional scripting checks such that cross-site scripting and SQL injection vulnerabilities might be discovered.
- The test team commenced a practice of writing and executing additional security tests, specifically in the areas of input validation, such as cross-site scripting and SQL injection.

We examined the Red Hat IT engineers' sentiments via the previously mentioned survey. Figure 2 compares the team's answers after they'd completed a tutorial with their answers after participating in three Protection Poker sessions. The survey results and our experience with the team yielded the following observations:

- *Software security knowledge*. As stated earlier and substantiated in Figure 2a, the team exhibited a range of knowledge about software security. We hypothesize that Protection Poker can raise a team's overall software security knowledge. However, early Protection Poker discussions can also raise individuals' awareness about how much they don't know about security. Hence, Figure 2a "post-use" demonstrates a wider spread of knowledge levels than does "post-tutorial."
- *Spreading software security knowledge*. Figures 2b and 2c illustrate the team's sentiments about Protection Poker's potential as a software security education practice and for spreading software security throughout the team.
- *Software security issues*. Figure 2d substantiates that the team discusses key software security issues during Protection Poker sessions.

Because Protection Poker is designed to elicit healthy discussions among everyone on the team, we empirically examined each member's discussion con-

tributions using two methods: contribution tallying and sampling. For contribution tallying, we kept a running score of how many contributions each person made to the discussion. We defined a contribution as any new idea brought to the group, whether trivial or significant—for example, a group member might say, "We need to protect this database because it's used by many other services as well," or "Compromising this system means leaking customer information." We didn't count side discussions and quick acknowledgments. The Red Hat IT team averaged 66 contributions for sessions that lasted 45 to 60 minutes. Figure 3 shows the contribution breakdown for each team member in a single session. A perfectly even contribution for nine people is roughly 11.1 percent per person, which many team members were close to (including the manager), demonstrating that Protection Poker provides a structure for input on software security from the whole team.

Although members made evenly distributed contributions, a single contribution's length often varied from person to person. Some people would make a single point at once and talk at length; others would speak rapidly, introducing multiple ideas. To estimate the relative time each person spent talking, we logged who was speaking every 30 seconds during the Protection Poker session. The sampling covers all conversation during this time, including moments when the manager is setting up the next vote or one person is describing the particular requirement to the team. Figure 3b shows the estimated breakdown of talk time per person during a single Protection Poker session. Again, the contributions were fairly evenly distributed. Interestingly, the manager made longer contributions (sometimes he would introduce a complex requirement to the group prior to discussion), and other members (such as Ethan) made many short contributions. In Ethan's case, all of his contributions were shorter than 30 seconds and never coincided with our sampling, so his estimated contribution percentage was zero. These results also indicate that Protection Poker provides a structure for the whole team to provide input on software security.

Our participation with the Red Hat team revealed some lessons learned relative to Protection Poker. First, teams shouldn't get discouraged if early Protection Poker sessions take longer than expected. Ultimately, the team spent a little more than one hour discussing the security implications of approximately 10 requirements, which is perhaps 6 to 8 minutes per requirement on average. We felt that no Protection Poker discussion was inefficient due to the amount of software security knowledge transfer that occurred in the meeting. Over time, though, the team will get more efficient as they get used to each other, build up a knowledge base of previous decisions on asset value, and better

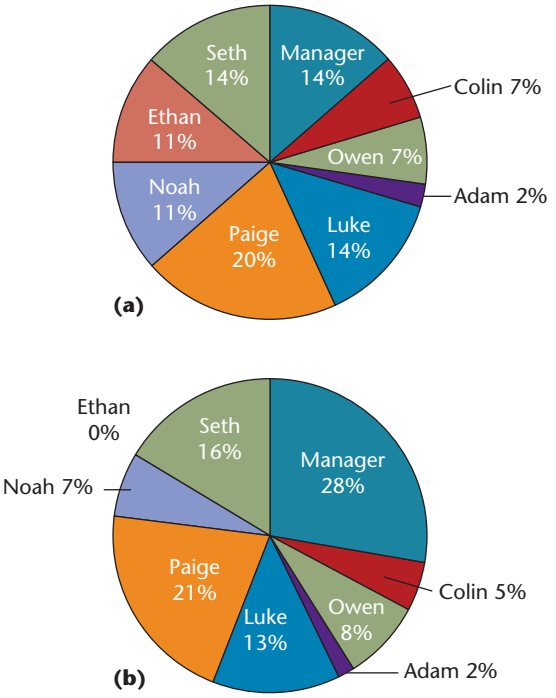

Figure 3. Contribution breakdown. We looked at (a) the percentage of Protection Poker discussion contributions and (b) the estimated percentage of time talking during the same Protection Poker session. (Note that names have been changed, but genders are preserved.)

understand the security risks. As with all meetings, the moderator must keep the discussion progressing.

Protection Poker voting provides a structure through which passive, quiet personalities have a chance to speak up. The longer the time between votes, the less likely a passive personality will speak up. The Red Hat team voted every three to five minutes.

Using Protection Poker should reduce vulnerabilities in the product through an overall increase of software security knowledge in the team. We observed four major benefits to using the Protection Poker practice:

- Security risk estimate and ranking. Albeit based on relative estimates, Protection Poker quantifies software security risk, which a team can then use to rank requirements. This ranking can help developers plan explicit actions to reduce security risk. The extended team obtains estimates via all members' expert opinions. Incorporating these opinions leads to improved estimation accuracy,[9,10] particularly over time.
- *Adaptation of requirement to reduce security risk.* The initial requirement might not reflect the need for securi-

ty functionality, such as role-based access or logging. Through the extended team's think-like-an-attacker brainstorming, these needs could surface, and the team can update the requirement accordingly.

- *Proactive security fortification to reduce security risk*. Teams who don't consider security issues as they develop the software might realize too late that they didn't allocate enough time in the development schedule to build a secure product, sometimes resorting to shipping a knowingly insecure one. Through Protection Poker, before requirement implementation begins, the extended development team has a chance to brainstorm and decide what explicit actions are necessary to reduce security risk, such as conducting a security inspection or intense input-injection testing for a Web form. The team can plan these explicit actions into the implementation schedule.
- *Software security knowledge sharing*. Protection Poker inspires a structured discussion of security issues that incorporates the extended development team's diverse perspectives. This discussion improves the team members' knowledge and awareness of what is required to build security into a product.

As evidenced by the case study and the benefits it brought to the Red Hat IT team, Protection Poker shows promise to improve not only software security but also the entire development team's security knowledge. Industrial teams who are interested in trying the Protection Poker practice should contact us for additional resources and with questions. We're interested in conducting a comprehensive empirical study with an industrial team. □

Acknowledgments

This research has been supported by the Center for Advanced Computers and Communications and 0716176 and the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by the NCSU Secure Open Systems Initiative (SOSI). We also thank Michael Gegick for his suggestions on the Protection Poker practice and his involvement in the Protection Poker feasibility study conducted with undergraduate students at North Carolina State University.

References

1. L. Williams, M. Gegick, and A. Meneely, "Protection Poker: Structuring Software Security Risk Assessment and Knowledge Transfer," *Proc. Int'l Symp. Engineering Secure Software and Systems* (ESSoS 09), Springer-Verlag, 2009, pp. 122–134.
2. I. Alexander, "On Abstraction in Scenarios," *Requirements Eng.*, vol. 6, no. 4, 2002, pp. 252–255.
3. M. Howard and S. Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.
4. G. Stoneburner, A. Goguen, and A. Feringa, "NIST Special Publication 800-30: Risk Management Guide for Information Technology Systems," Nat'l Inst. Standards and Technology, #800-30, July 2002.
5. M. Cohn, *Agile Estimating and Planning*, Prentice-Hall, 2006.
6. K. Schwaber and M. Beedle, *Agile Software Development with SCRUM*, Prentice-Hall, 2002.
7. B. Boehm, *Software Risk Management*, IEEE CS Press, 1989.
8. G. McGraw, *Software Security: Building Security In*, Addison-Wesley, 2006.
9. N.C. Haugen, "An Empirical Study of Using Planning Poker for User Story Estimation," *Proc. Agile 2006*, 2006, p. 9 (electronic proceedings).
10. K. Moløkken-Østvold and N.C. Haugen, "Combining Estimates with Planning Poker—An Empirical Study," *Australian Software Eng. Conf.* (ASWEC 07), Elsevier Science, 2007, pp. 349–358.

Laurie Williams *is an associate professor in the computer science department at North Carolina State University. Her research interests include software security, reliability, testing, and process. Williams has a PhD from the University of Utah and an MBA from Duke University. She's a member of IEEE and the ACM. Contact her at williams@csc.ncsu.edu.*

Andrew Meneely *is a PhD candidate in computer science at North Carolina State University. His research interests include software security, reliability, metrics, and testing. Meneely has an MS in computer science from North Carolina State University. Contact him at apmeneel@ncsu.edu.*

Grant Shipley *is a manager of software development at Red Hat. His interests include finding more efficient ways of developing software in the open source community. Shipley has more than 10 years of software engineering experience working with Java in Linux environments. He's a certified Scrum master. Contact him at gshipley@redhat.com.*