Randomizing Regression Tests Using Game Theory

Nupul Kukreja, William G.J. Halfond, Milind Tambe
University of Southern California
Los Angeles, California, USA
Email: {nkukreja, halfond, tambe}@usc.edu

***Abstract*—As software evolves, the number of test-cases in the regression test suites continues to increase, requiring testers to prioritize their execution. Usually only a subset of the test cases is executed due to limited testing resources. This subset is often known to the developers who may try to "game" the system by committing insufficiently tested code for parts of the software that will not be tested. In this new ideas paper, we propose a novel approach for randomizing regression test scheduling, based on Stackelberg games for deployment of scarce resources. We apply this approach to randomizing test cases in such a way as to maximize the testers' expected payoff when executing the test cases. Our approach accounts for resource limitations (e.g., number of testers) and provides a probabilistic distribution for scheduling test cases. We provide an example application of our approach showcasing the idea of using Stackelberg games for randomized regression test scheduling.**

I. Introduction

The size of regression test suites continues to grow as software evolves. Often there are more test cases to execute than is feasible given schedule constraints. Therefore, project teams must prioritize the regression test cases to be executed. In many cases, developers can easily anticipate the test cases that will be run, since this is based on widely disseminated requirement priorities. As a result the developers can commit insufficiently tested code into the version control repository, fully aware that a particular test case may not be executed for a given schedule – following a "code-now-fix-later" approach. A developer's motivation to do this can be high if there is a significant risk or penalty for missing the delivery deadline of the software system.

Existing regression prioritization schemes generally lead to deterministic testing activities. Developers are able to predict which test cases will be executed and when. Even in the case of automated testing, only a prioritized subset of the tests may be executed periodically to get quick feedback on the functionality of the system. Complete regression test suites may be run less frequently e.g., every alternate weekend or at the end of a particular iteration. This determinism allows the developers to "game" the system by checking-in insufficiently tested code and fixing it either after the execution of the test cases or just prior to their scheduled execution. In case of low priority test cases, the turn-around time between the initial commit of the code and fixing the errors may be much higher. Random test scheduling would help to address this problem, but ignores the fact that some test requirements are more important than others. Adding priority weights to the random testing is another improvement, but even this somewhat predictable distribution would allow developers to find a sweet spot that balances a requirement's extra testing work with the likelihood of their code being tested.

This tension between testers and developers is reflected in many problems addressed in the game theory community. In particular, a *security game* models a situation where limited security resources (defenders) must be deployed for protecting public infrastructure such as ports, trains, airports etc. These games assume that an adversary (attacker) can perform perfect surveillance and exploit any predictable patterns in the security schedule before planning an attack. Thus, it is in the defender's best interest to randomize the allocation of security resources to decrease the predictability in scheduling, making it difficult for an attacker to plan an attack. The game solution seeks to optimize the defender's payoff given resource limitations and in face of an adaptive adversary.

In this new ideas paper we introduce the analogous idea of a *testing game*. In this type of game the testers act as defenders guarding against loss of software quality and developers assume the role of attackers who may check in insufficiently tested code that could decrease software quality. Given a regression test suite, computing the solution to this game allows testers to select an optimal test case distribution. The method of computing this distribution takes into account both the developers' incentives to game the system and the testers rewards and penalties for testing or not testing the code. The probability of executing a particular test case is therefore proportional to the overall value of the requirement and effectiveness of the test case.

The paper is organized as follows: In Section II, we introduce basic information about game theory and security games. We outline our approach and map it to the concepts of a security game in Section III. To illustrate the viability of such a technique, we present a small analytic based case study in Section IV. We discuss related work and conclude in Sections V and VI.

II. Background

Our testing game is derived from the idea of a security game, a well-known model in the game theory community.

A. Game Theory

Game theory [16] is a study of strategic decision making – an abstract mathematical theory for analyzing interactions among multiple intelligent players. These players may be people, corporations, software agents or as in our case, testers

TABLE I: Security game: Defender vs. Adversary. Defender payoffs lower-left, adversary upper right

		Adversary	
		Terminal 1	Terminal 2
Defender	Terminal 1	-3	1
		5	-1
	Terminal 2	5	-1
		-5	2

and developers. In the context of security games, one of the players is usually part of the security forces e.g., federal air marshals, police etc., with the adversaries as the other player. Game-theoretic approaches assume that the players will anticipate each others actions and act appropriately.

Mathematically a game is represented as a matrix (i.e., normal form) showing the strategies available to each player and their corresponding payoffs. For example, consider an airport with two terminals, 1 and 2, a single police unit to protect them, and one adversary. Table I shows the matrix corresponding to this game. The possible police actions are shown across the rows and those of the adversary across the columns. The numbers in the matrix represent payoffs for each player. For this example we assume the payoffs range from -5 to 5. The police can protect either terminal 1 or terminal 2 and an adversary can attack either of the terminals (i.e., their *strategies* when playing the game). In a simultaneous move game, both players play at the same time and cannot observe the action of the other player before choosing their response. However, in the context of a security game, the defender commits to a strategy that is fully observable by the adversary before responding with their strategy. For the game in Table I, assume that terminal 1 is more important than terminal 2 and the police choose to always protect terminal 1. The adversary can observe this after conducting necessary surveillance and choose to attack terminal 2. Since there are no police at terminal 2, we assume the attack succeeds (with a positive payoff). If the police choose to protect terminal 2, the adversary can attack terminal 1 for a positive payoff.

Thus, if the police commit to a deterministic strategy of always protecting either terminal 1 or terminal 2, the adversary can exploit the predictability of the protection schedule and choose to attack the other unprotected terminal. This results in a positive payoff for the adversary and a corresponding negative payoff for the defender. If however, the police were to employ a *mixed strategy* i.e., randomize their actions, and 60% of the time protect terminal 1 and 40% of the times protect terminal 2, it would lead to a higher expected payoff for the police. The randomization seeks to optimize the defender's payoff in face of an adaptive adversary and is proportional to the payoffs of both players. We assume the adversary can perfectly observe the probabilistic distribution of the protection schedule (60%–40% split between terminals) but precisely which terminal will be protected on the day of the planned attack remains unknown. This increases adversary uncertainty thereby increasing the expected reward for the defender.

These types of games are called Stackelberg games [17] where the defender commits to a strategy that is fully observable by the adversary before choosing his/her response. In the above security game the police has effectively committed itself to a mixed strategy (60%–40% split between the terminals). The challenge for game theory models is to ensure that the 60%–40% split of security resources between the terminals is indeed optimal or determine if there is another optimal split. Solving this optimization problem for large games requires efficient computational algorithms [14].

B. *Security Games*

Security games are based on the principle that what is good for the adversary is bad for the defender and vice versa. However, most security games are non-zero sum (i.e., sum of all payoffs is not zero) [11]. This is because an adversary views some targets as highly valuable and they may not be of equal value to the security forces. An adversary may even view a failed attack as a success owing to the publicity and fear generated as a result. In a security game, if an adversary attacks a target protected by the defender, then the adversary has a worse (i.e., lower) payoff than attacking it when it is unprotected. The situation is reversed for the defender i.e., the defender has a higher payoff if the target is attacked when protected than when it's unprotected. Given such security games it is vital to find the optimal allocation of security resources that will maximize the defenders expected payoff. This involves computing what is known as the game's *Strong Stackelberg Equilibrium (SSE).*

A well known solution concept in game theory is Nash Equilibrium, which computes an optimum strategy for each player such that no player has any incentive to deviate to another strategy. Stackelberg equilibrium is a refinement of Nash equilibrium specific to Stackelberg games where each player chooses a best-response (i.e., Nash equilibrium) in any subgame (i.e., partial sequence of actions) of the original game. Thus, a player observes the response of the other player and responds with an optimum action from his/her set of possible actions. There are two types of unique Stackelberg equilibria – strong and weak [2]. The strong form assumes that in case of ties (i.e., multiple best responses) the adversary will break ties in favor of the defender i.e, select a response that is optimal for the defender. Whereas the weak form assumes that the adversary will select the worst response for the defender. A SSE exists in all Stackelberg games but a weak one may not [1]. Thus, security games solve for an SSE due to this crucial existence result.

III. Testing Game

Software test scheduling can be viewed as a *security game* between testers and developers. Testers commit to a testing strategy i.e., executing the most valuable test cases, given resource/time constraints, to ensure high software quality. Thus, testers act as defenders guarding against the loss of software quality. Developers play the role of attackers who may check-in insufficiently tested code that could lead to bugs in the system, potentially decreasing software quality. The testers must ensure that errors are caught and fixed prior to transitioning the software system into production. We assume that developers prefer, with some probability, to check in code and get credit for completing application functionality sooner rather than spending additional effort on sufficient testing.

We cast regression test scheduling as a Stackelberg game where the testers commit to a randomized test strategy fully observable by the developers. That is, the developers know the probability of execution of each test case. However, when each particular test case will be executed is unknown. We then solve this game to compute a strong Stackelberg equilibrium (SSE) that provides the optimal probability distribution of the test cases to be selected for a randomized schedule. The distribution, in essence, prioritizes over the regression test suite that's been selected.

We explain our approach in four steps: *Step 1:* We define the test case utilities for the tester and developer (Section III-A). *Step 2:* We define their expected payoff functions given some probability distribution of the test cases (Section III-B). *Step 3:* We then show how to *solve* this game in order to maximize the expected payoff for the testers (i.e., computing SSE). The solution of the game provides the optimum probability distribution of the test cases (Section III-C). *Step 4:* Finally, we randomly sample from the probability distribution of the test cases to generate a test schedule (Section III-D).

A. Computing Test Case Utilities for Tester/Developer

In a game each player has an incentive (or penalty) for taking a particular action. The incentive or penalty is referred to as the utility of the player and is represented by a positive or negative number, respectively. These utilities are indicative of the relative importance of the particular action for that player. In a testing game, testers are assumed to have a positive utility for executing a test suite for testing a particular requirement (i.e., potential value of catching a bug) and a negative utility otherwise (i.e., potential loss of missing a bug). These utility scores are proportional to the value of the requirement and can be derived from the requirement priority itself or from expert judgment as is done in Planning Poker [4], the Delphi method [9], and other requirement prioritization schemes [7]. If a bug is caught by the tester, the developer gets a negative utility since they incur a penalty of fixing it and possibly interrupting what they were working on. The developer gets a positive utility if the insufficiently tested code is not tested by the tester since there is no penalty for fixing a bug, if any.

For our game model we assume that the test cases are grouped by requirements i.e., each requirement has one or more test cases covering it that may be grouped into individual test suites. A single test suite may cover multiple requirements but for simplicity we assume that each test suite corresponds to exactly one requirement – the mapping of test suites to requirements and vice versa is usually maintained in the software engineering process to enable traceability. Let $T = \{t_1, \ldots, t_n\}$ be the set of test cases pertaining to each requirement. The testers have a set of resources–either human testers or dedicated machines for automated testing–that execute test cases *covering* each requirement. We then capture the tester's and developer's utilities, on a -10 (high penalty) to $+10$ (high value) scale, for each requirement as shown in Table II. The tester's utilities for an uncovered requirement (i.e., test cases covering the requirement are not executed) is denoted by $U^u_\tau(t)$ and for a covered requirement is denoted by $U^c_\tau(t)$. Similarly, $U^c_d(t)$ and $U^u_d(t)$ are the developer's utilities when checking in insufficiently tested code for covered and uncovered requirements as mentioned above. These utilities are proportional to the overall priority of the requirement. For example, for requirement 5 in Table II, $U^c_\tau(t) = 9$ is the tester's utility for executing a test case covering that requirement and $U^u_\tau(t) = -9$ is the utility for not executing that test case. Similarly, $U^c_d(t) = -10$ is the developer's utility if the insufficiently tested code is caught by the test case and $U^u_d(t) = 3$ is their utility if the insufficiently tested code is not caught. These utility numbers would be representative of a critical 'must-have' requirement. The tester gets a high utility for testing the requirement, since it is mandatory to have that requirement, and a high penalty if it is not tested since a bug could prevent the users from accessing the core functionality of the system and lead to monetary losses for the organization. Also, if the developer checks in insufficiently tested code for requirement 5 and a bug is detected by the tester, they have a high penalty since it may involve substantial rework to get the system up and running quickly. However, the developers have low positive utility for checking in insufficiently tested code owing to the critical nature of the requirement.

B. Computing Tester/Developer Payoffs

In a security game, defenders may employ mixed strategies i.e., selecting a particular action with some probability as opposed to a deterministic choice. This use of a distribution decreases the predictability that may be exploited by an adversary. For our testing game we refer to the set of distributions as a *coverage vector*, C, that gives the probability, c_t, that a particular test suite, t, covering a particular requirement will be executed. We then compute the expected payoffs for the testers and developers as shown in Equations (1) through (3)

$$U_\tau(t, C) = c_t \cdot U^c_\tau(t) + (1 - c_t) \cdot U^u_\tau(t) \tag{1}$$

$$U_\tau(C, R) = \sum_{t \in T} r_t \cdot (c_t \cdot U^c_\tau(t) + (1 - c_t) \cdot U^u_\tau(t)) \tag{2}$$

$$\Gamma(C) = \{t : U_d(t, C) \geq U_d(t', C) \, \forall t' \in T\} \tag{3}$$

Equation (1) gives the expected payoff for the tester for an individual test suite, t, given C, when the developer checks in code. The same notation is used for computing payoffs for the developers (not shown), replacing U_τ with U_d. We also define the notion of a *requirement vector, R*, that gives the probability of checking in insufficiently tested code, r_t, for a particular requirement. For simplicity, we restrict the requirement vector

TABLE II: Compact representation of game showing tester and developer utilities

	Requirement 1		Requirement 2		Requirement 3		Requirement 4		Requirement 5	
	Covered	Uncovered	Covered	Uncovered	Covered	Uncovered	Covered	Uncovered	Covered	Uncovered
Tester's utility	2	-10	7	-4	6	-1	9	-9	9	-9
Developer's utility	-7	4	-1	3	-6	5	-3	7	-10	3

to *attacking* a single requirement with the probability of 1. That is, the developer checks in insufficiently tested code for a single requirement at a time with the probability of 1. This assumption simplifies computing the solution of the game (i.e., computing the SSE) without loss of generality. The tester's total expected payoff, given coverage and requirement vectors, is given by Equation (2). It is the sum of the payoffs for each individual test suite times the probability, r_t, of the developer attacking the corresponding requirement. Equation (3) defines a *requirement set*, $\Gamma(C)$, which contains all the requirements that yield the maximum expected payoff for the developer given coverage vector, C. That is, the set of requirements for which the developer is most likely to check in insufficiently tested code.

C. Maximizing Tester's Expected Payoff – Computing SSE

The aim of each player in a game is to maximize their individual payoffs. In the testing game we solve for the optimum coverage vector, C, that maximizes the testers' total expected payoff. Testers then commit to the randomized test case scheduling strategy given by C. Test cases are grouped by requirements (Table II), which are known to both testers and developers and thus both players are aware of each others strategies. As per the Stackelberg formulation of the game, we assume the developers can perfectly observe the testing strategy and deduce the probability of execution of the test suite for the corresponding requirement. The developers then intend to maximize their utility by selecting the best response to the tester's strategy. In this section we detail the algorithm used for solving the testing game to obtain C that maximizes the tester's utility–the SSE of the game.

Most security games can be expressed and solved efficiently using the compact representation shown in Table II, as opposed to the normal form representation in Table I. For solving our testing game, we use the ERASER (Efficient Randomized Allocation of SEcurity Resources) algorithm [5], which is reproduced below.

$$\max \delta \tag{4}$$

$$r_t \in \{0,1\} \qquad \forall t \in T \tag{5}$$

$$\sum_{\forall t \in T} r_t = 1 \tag{6}$$

$$c_t \in [0,1] \qquad \forall t \in T \tag{7}$$

$$\sum_{\forall t \in T} c_t \leq m \tag{8}$$

$$\delta - U_\tau(t,C) \leq (1-r_t)\cdot Z \qquad \forall t \in T \tag{9}$$

$$0 \leq k - U_d(t,C) \leq (1-r_t)\cdot Z \qquad \forall t \in T \tag{10}$$

It takes a compact representation of a *security game* as input and solves for an *optimal* coverage vector for the *defender* (i.e., the tester in our case). The algorithm is a mixed-integer linear program (MILP) as shown by Equations (4) through (10).

Equation (5) restricts the requirement vector probabilities to 0 or 1 and Equation (6) constrains the vector to only attack a single requirement, as explained previously. Equation (7) restricts the coverage vector probabilities in the range $[0,1]$ and Equation (8) constrains the coverage by the number of available resources, m. In Equations (9) and (10) Z is an arbitrarily large constant, greater than the maximum utility of the tester/developer. Equation (9) defines the testers expected payoff subject to the requirement *attacked* by the developers, as given by the requirement vector, R (i.e., insufficiently tested code for the requirement is committed). Equation (9) places an upper bound $U_\tau(t,C)$ on δ, but only for the *attacked* requirement. For all other requirements the right hand side is arbitrarily large (equal to Z). Since, the objective function maximizes δ (tester's utility), for any optimal solution $\delta = U_\tau(C,R)$ (from (2) and (4)) it also implies that C is maximal, given R for any optimal solution. Similarly, Equation (10) forces the developer to select a requirement in the requirement set, $\Gamma(C)$. The first part of the constraint $k - U_d(t,C) \geq 0$ implies that k must be at least as large as the maximal payoff for attacking any requirement. The second part forces $k - U_d(t,C) \leq 0$ for any requirement that is attacked in R. The constraint is violated if the requirement vector specifies a requirement that is not maximal. Equations (9) and (10) together imply that C and R are mutual best responses in any optimal solution. An optimal solution to the ERASER MILP thus corresponds to an SSE of the testing game [5].

D. Test case scheduling

Solving the testing game shown in Table II with a single testing resource (i.e., setting $m = 1$ in Equation (8)) provides the coverage probabilities as shown in Table III . The probability distribution of test cases is proportional to the value of the requirements for both players and not just for the tester or developer. Solving the testing game takes into account both tester's and developer's utilities to compute the optimum distribution. The distribution is effectively a weighted prioritization of the test cases in face of an adaptive adversary i.e., the developer. The test suites are then sampled from this distribution to create a randomized test schedule. We need to recompute coverage probabilities only when a new requirement is added or an existing one is removed, since test suites are grouped by the requirement they belong to.

TABLE III: Coverage probabilities

c_1	c_2	c_3	c_4	c_5
0.1398	0.1344	0.2307	0.4538	0.0414

Thus, updating the code or test cases does not incur the cost recomputation of the coverage probabilities.

The distribution in Table III may seem counter-intuitive. For example, in Table II, Requirement 4 and 5 have the same payoffs for the tester but radically different distributions. The corresponding test suite for requirement 4 has the highest probability of execution since it's also the one where the developers are most likely to check in insufficiently tested code as indicated by their utilities. Although requirement 5 has the same utilities for the tester, it has the lowest probability of execution given the utilities of the developer. The probability distribution is computed by considering the utilities and payoffs of both, testers and developers. As a result the distribution, seemingly counter-intuitive at first is actually optimal [15] in a two player game setting.

The equations we defined above are able to handle multiple testing resources e.g., more human testers or machines. Separate ERASER instances can be executed for calculating the coverage vector specific to the particular resource. For parallel test case execution, the number of testing resources, m in Equation (8), is increased accordingly. This leads to higher values of the coverage probabilities. That is, the more resources that are available, the more testing that can be done.

IV. Evaluation

To provide preliminary validation of our approach, we perform a small case study of its ability to generate test suite distributions with higher tester payoff. In this case study we compare the tester's payoff associated with using our approach against a uniform random approach. The expected payoffs for the tester and developer, given the coverage probabilities previously computed by our approach (Table III), are shown in Table IV. Table V shows the expected payoffs for the tester and developer if the test cases are distributed uniformly at random (i.e., $c_t = 0.2\,\forall t \in T$).

TABLE IV: Expected Payoffs - Our Approach

	t_1	t_2	t_3	t_4	t_5
Tester	-8.32	-2.52	0.61	-0.83	-8.26
Developer	2.46	2.46	2.46	2.46	2.46

TABLE V: Expected Payoffs - Uniform Random

	t_1	t_2	t_3	t_4	t_5
Tester	-7.6	-1.8	0.4	-5.4	-5.4
Developer	1.8	2.2	2.8	5	0.4

The expected value for each test case, t_i, is computed using Equation (1). As an example, we compute the expected payoff for the tester $U_\tau(1, C)$ and developer $U_d(1, C)$, given the distribution of test case 1 (Table III) for requirement 1 (Table II) as follows:

$$U_\tau(1, C) = 0.1398 \times 2 + (1 - 0.1398) \times -10 = -8.32$$
$$U_d(1, C) = 0.1398 \times -7 + (1 - 0.1398) \times 4 = 2.46$$

The results in Tables IV and V are similarly computed for each of the test cases. In our approach the expected payoffs for developers is the same for each requirement. As mentioned previously, in an SSE the developer breaks ties in favor of the tester. Thus, the requirement vector, $R \equiv \{0, 0, r_3 = 1, 0, 0\}$ as $t_3 = 0.61$, is optimal for the tester, as seen in Table IV. In an SSE the payoff for the developer is strictly greater than the utility of *attacking* any other uncovered requirement. (By definition, since SSE consists of mutual best responses for both, tester and developer.) Thus, the ERASER algorithm sets the expected payoff for each test case, for the developer in the requirement set $\Gamma(C)$, equal to that of the SSE. Hence, the expected payoffs for the developer are the same for all test cases. Test case distributions are then computed in order to maintain a level of *indifference* for the developer in attacking a particular requirement from the requirement set. That is, we compute a test case distribution, c_t, by making it equally attractive for the developer to check in insufficiently tested code for each requirement.

The uniform random distribution of test cases doesn't take into account the developer's utilities and payoffs into consideration. It's value neutral since it assumes each test case to be equally important for scheduling. For example, the probability of executing the test case for requirement 4, computed by our approach ($c_4 = 0.4538$) is much higher than 0.2. This implies that test case 4 will be scheduled almost twice as often as opposed to the uniform distribution. This takes into account that the developer is most likely to gain by checking in insufficiently tested code for requirement 4 than any other requirement and thus has a higher probability of execution. The developer utilities are ignored in the case of uniform distribution and all test cases are treated as being equally valuable. The ERASER algorithm effectively computes the optimum distribution that maximizes the tester's expected payoff [15]. Also, as an adversary will act rationally, the developer too will get a higher expected payoff (Table IV) with our approach. Thus, prefering that over the uniform random approach (Table V).

Our case study shows that the expected payoff is higher in case of our approach as opposed to a uniform random distribution. However, we are limited in the generalizability of our case study results. In future work we plan to include sensitivity analysis with larger datasets and more realistic data. However, our results at this stage indicate that the use of our approach can improve tester payoff against a baseline approach.

V. Related Work

We provide a brief survey of the application of game theory to software testing along with a discussion of several test case prioritization approaches. We highlight the key differences between the related work and our approach.

Feijs [3] models software testing as a strategic game and shows its equivalence to the Prisoner's Dilemma [10]. Feijs focuses on understanding the interactions among developers and testers using game theoretic principles. However, the analysis is very coarse grained and studies the interactions from a project management point of view. Similarly, Yilmaz and O'Connor [19] look at maximizing the value of software development using game theory. The paper focuses on creating economic mechanisms to align the motivations of the various competing stakeholders to ensure delivery of a high value software system. As such, it covers the entire software development lifecycle and does not focus on techniques specific for software testing. There has also been extensive research on prioritizing test cases for regression testing ([18], [20], [13], [6]) to decrease the time/effort overhead when running large regression test suites. However, these approaches do not take into account the possibility of developers gaming the system by checking in untested code given a predictable testing schedule (i.e., the game theory aspect of software testing.) Value-based testing practices [12] also fail to take this into account and focus on managing/prioritizing testing activities to maximize business value. Li [8] proposes creating test schedules by sampling a subset of test cases uniformly at random. Its primary focus is predicting test coverage by random sampling within some confidence intervals.

VI. Conclusion

In this paper we proposed a novel application of game theory and security games to software testing. In our approach we use game theoretic principles to compute test case distributions based on test case priority that takes into account both developers' and testers' motivations and resource limitations. Our approach weights the test cases with respect to the utilities of the two players, testers and developers, and computes a distribution that maximizes tester payoff. This distribution of test cases is optimum, given the utilities of both players. In a small case study we showed that the expected payoff for testers is higher in the case of our approach as opposed to a uniform random distribution. This provides preliminary support to our idea that casting regression testing as a Stackelberg game can help increase software quality.

Acknowledgments

This research is supported by MURI grant W911NF-11-1-0332 and NSF award 1321141.

References

[1] T. Basar, G. J. Olsder, G. Clsder, T. Basar, T. Baser, and G. J. Olsder. *Dynamic noncooperative game theory*, volume 200. SIAM, 1995.

[2] M. Breton, A. Alj, and A. Haurie. Sequential stackelberg equilibria in two-person games. *Journal of Optimization Theory and Applications*, 59(1):71–97, 1988.

[3] L. Feijs. Prisoners dilemma in software testing. *7e Nederlandse Testdag*, page 65, 2001.

[4] J. Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, 3, 2002.

[5] C. Kiekintveld, M. Jain, J. Tsai, J. Pita, F. Ordóñez, and M. Tambe. Computing optimal randomized resource allocations for massive security games. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 689–696. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[6] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129, 2002.

[7] N. Kukreja, B. Boehm, S. S. Payyavula, and S. Padmanabhuni. Selecting the most appropriate framework for value based requirements prioritization.

[8] W. Li and M. J. Harrold. Using random test selection to gain confidence in modified software. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 267–276. IEEE, 2008.

[9] H. A. Linstone and M. Turoff. *The delphi method*. Addison-Wesley Reading, MA, 1975.

[10] W. Poundstone and N. Metropolis. Prisoner's dilemma: John von neumann, game theory, and the puzzle of the bomb. *Physics Today*, 45:73, 1992.

[11] R. Powell. Defending against terrorist attacks with limited resources. *American Political Science Review*, 101(3):527, 2007.

[12] R. Ramler, S. Biffl, and P. Grünbacher. Value-based management of software testing. In *Value-Based Software Engineering*, pages 225–244. Springer, 2006.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[14] M. Tambe. *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*. Cambridge University Press, 2011.

[15] M. E. Taylor, C. Kiekintveld, and M. Tambe. Evaluating deployed decision support systems for security: Challenges, analysis, and approaches.

[16] J. Von Neumann and O. Morgenstern. *Theory of games and economic behavior (commemorative edition)*. Princeton university press, 2007.

[17] H. Von StackelberG. *Market structure and equilibrium*. Springerverlag Berlin Heidelberg, 2011.

[18] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[19] M. Yilmaz and R. V. O'Connor. Maximizing the value of the software development process by game theoretic analysis. In *Proceedings of the 11th International Conference on Product Focused Software*, pages 93–96. ACM, 2010.

[20] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.