Toward Effective Adoption of Secure Software Development Practices

Shams Al-Amin[†], Nirav Ajmeri[‡], Hongying Du[‡], Emily Z. Berglund[†], Munindar P. Singh[‡]

[†]*Department of Civil, Construction, and Environmental Engineering,* [‡]*Department of Computer Science, North Carolina State University, Raleigh, NC 27695*

Abstract

Security tools, including static and dynamic analysis tools, can guide software developers to identify and fix potential vulnerabilities in their code. However, the use of security tools is not common among developers. The goal of this research is to develop a framework for modeling the adoption of security practices in software development and to explore sanctioning mechanisms that may promote greater adoption of these practices among developers. We propose a multiagent simulation framework that incorporates developers and manager roles, where developers maximize task completion and compliance with security policies, and the manager enforces sanctions based on functionality and security of the project. The adoption of security practices emerges through the interaction of manager and developer agents in time-critical projects. Using the framework, we evaluate the adoption of security practices for developers with different preferences and strategies under individual and group sanctions. We use a real case study for demonstrating the model and initialize the occurrence of bugs using a 13 year database of bug reports for the Eclipse Java Development Tools. Results indicate that adoption of security practices are significantly dictated by the preferences of the developers. We also observed that repetitive sanctions may cause lower retention of developers and an overall decrease in security practices. The model provides comparison of security adoption in developers with different preferences and provides guidance for managers to identify appropriate sanctioning mechanism for increasing the adoption of security tools in software development.

Keywords:

Multiagent system, security practices, adoption, decision theory, sanction

1. Introduction

Secure software development tools, or security tools, are programs that analyze software to help developers find and fix vulnerabilities [1, 2]. Such tools may analyze software code, find vulnerabilities [3, 4], warn developers of probable violations of coding standards [5], or find programming errors through static (FindBugs) [6] or dynamic analysis (Valgrind plugins) [7]. To illustrate the functionality and advantages of a static security analysis tool, consider FindBugs as an example. FindBugs can be run as a plug-in for the Eclipse and NetBeans integrated development environments, using Ant or Maven, from the command line or as a separate tool on its own [1, 8, 9]. FindBugs can group each bug pattern into a category of correctness, bad practice, performance, or internationalization, and prioritizes bugs as high, medium, or low. Findbugs also offers a few fixes. However, despite the advantages of these tools, the use of security tools is not common among developers [2]. In a survey [10], 60% developers responded that in their organization developers run FindBugs in an ad-hoc way, 80% responded that there is no policy on how soon each FindBugs issue must be human-reviewed, and 83% responded that FindBugs warnings are not inserted into a separate bug-tracking database in their organization.

Adoption of security practices. Previous research that looked into the adoption of security practices are based on quantitative data collected through surveys. Witschey et al. [2] collected quantitative data on the relative importance of factors through an online survey of software developers. They identified 39 factors that affect adoption. These are social system factors such as security concern and awareness, policies and standards, structures, education and training, and culture; innovative factors such as relative advantage, observability, complexity, and trialability; communication channel factors such as trust and exposure; and potential adopter factors. They built a combined logistic regression model using the 39 factors and found six factors are statistically significant. The six significant factors are observability, advantages, policies, inquisitiveness, education and exposure. Kina et al. [5] analyzed the decision criteria of software developers based on prospect theory and concluded that developers avoid selecting tools if the probability of

the effect of the tools is unknown. Araújo et al. [11] investigated the effectiveness of existing bug prediction approaches with procedural systems and compared their effectiveness using standard metrics, with adaptations when needed. Ayewah et al. [10] pointed out that the users' willingness to review warnings and fix issues also depends on the characteristics and organization of the project, the time investment they are willing to put into each review, and their tolerance for false positives.

Although surveys can help explore the factors influencing adoption, it is logistically challenging to observe how each developer works in real time. Simulation can explore and predict adoption patterns in different scenarios. Simulations based on technology acceptance model [12, 13], diffusion of innovation model [14, 15], and application of social network theories [16, 17] and decision theories [18, 19, 20] have been successfully used in modeling adoption and can be useful in understanding adoption of security practices in software development. Dignum and Dignum [21] proposed to use ideas from social practice theory to support reasoning about action and planning of intelligent agents in a social context. From a decision theory perspective, the factors that influence adoption of security practices can be viewed as a developer's individual preferences and the perceived utility of using security practices. While a developer will prefer using security tools only if it leads to a rationally rewarding outcome for his or her individual utility, a manager who oversees the adoption and the overall quality of the outcome may offer rewards or punishments, because the security of the end product depends on the security practices. We propose a rational decision making framework to explain the adoption of security practices by developers. The model simulates individual developers' preferences and decision making based on their perceptions of the advantages of security tools. The model also includes a manager who has full observability of the adoption practices and implements a sanction mechanism that enforces a policy to use security practices to meet a specified standard.

Norms and Sanction. We recognize the term, norms, to describe "directed normative relationships between participants in the context of an organization" [22, 23]. Norms are powerful means for regulating interactions among autonomous agents [24, 25, 26]. Social interactions form norms which are influential in dictating what behaviors are expected in a community and of the system [27, 28]. Failure to comply with normative expectations is met with a sanction, which is a consequence for norm violation applied to a principal

or group of principals, by a sanctioning agent. A sanction may be positive or negative and is manifested in reprimand or reward, respectively [29]. In the context of the developers' adoption environment, a norm violation would be met with a negative sanction. When an individual is singled out and censured for defecting against a norm, we recognize this as an *individual* sanction. Alternatively, when a sanction is applied to a group of individuals for actions of a subset of that group, we recognize this as a *group* sanction, also known as *collective* sanction [30].

Example 1. *Consider Alex, Barb, Charlie, and Dave, who are software developers working as a team to deliver a product. Erin is their manager. Erin divides the project into multiple tasks and assigns those to the developers. Erin wants to make sure the product is delivered on time and meets functionality and security requirements. Erin and her team use FindBugs to identify security bugs. Alex and Barb are experienced developers who use security tools. Charlie and Dave are new to the team and not aware of security practices.*

Consider a scenario, where Charlie and Dave learn to use FindBugs. Everyone follows the standard security practices and delivers the product on time. Once the product is launched, its functionality and security are found as satisfactory. Erin rewards her team and encourages everyone to continue following security practices.

Now consider a second scenario where the project is time-critical, and Charlie skips executing FindBugs to deliver the product just with required functionality. This can lead to multiple alternatives: (1) Once the product is launched, it is flagged for security concerns. Erin identifies that Charlie did not follow the standard security practices on the concerned artifacts and scolds him. Charlie realizes the importance of using security tools. (2) Based on the evaluation, Erin scolds her team to use security tools in future. (3) Alex finds out that Charlie is not following the security practices and prompts him to use security tools. The illustrative example can be used to distinguish among *individual*, *group* and *peer* sanctions. Erin sanctioning Charlie is an example of *individual sanction* where the manager monitors and sanctions individuals who do not follow a norm. Erin scolding everyone can be considered as a *group sanction* where the sanctions are imposed on a whole group regardless of who in the group violated or satisfied a norm. Alex prompting Charlie is an example of *peer sanction* where a peer sanctions another peer.

Multiagent Systems. Norms are used to regulate agent behavior and facilitate collaboration in open MASs [31, 32]. Savarimuthu and Cranefield [33] surveyed simulation models of norms in multiagent systems and proposed five phases of the norm life-cycle based on a socio-computational viewpoint. These are norm creation, identification, spreading, enforcement and emergence. Savarimuthu et al. [34, 35] further proposed a framework for social norm emergence in virtual agent societies. They argue that norms can be established through a bottom-up process based on a distributed, peer to peer punishment mechanism and demonstrate that the mechanism works on top of dynamically created networks. Dam and Winikoff [36] compare prominent agent-oriented methodologies based on an attribute-based framework which address four major areas of concepts, modeling language, process and pragmatics. Dam and Ghose [37] present a framework that supports designers in evolving software models in a collaborative modeling setting built upon the well-known Belief Desire Intention agent architecture. Meyer et al. [38] provide a general framework for multiagent context-sensitive merging and also investigate the link between such merging operations and the aggregation operations studied in social choice theory. Ghose et al. [39] outline a methodology for identifying the optimization norms that underpin other norms and then define a notion of compliance for optimization norms, as well as a notion of consistency and inconsistency resolution. Multiagent modeling has also been widely used in software development [40, 41, 42, 43, 44]. However, few studies [22] explore the adoption of security practices in software development. Our research fills this gap.

Contributions. The research addresses the following research question: *Which sanctioning mechanism promotes greater adoption of security practices?* We provide a model that simulates the adoption of security tools among developers. The model can be applied to identify appropriate sanctioning mechanism for increasing the use of security tools among a group of developers with heterogeneous skills. Model output demonstrates the emergence of the adoption and use of security tools by simulating the system dynamics as the interactions among developers and a manager in the completion of project tasks.

Organization. Section 2 describes the security practices adoption model. Section 3 details the simulation, and section 4 presents the experiments and their results. Section 4.5 discusses the limitations of the framework and threats to validity and section 5 concludes with important future directions.

2. Security Practices Adoption Framework

The framework simulates the dynamic interactions between developers and a manager in a time-critical project. Figure 1 shows the interactive modules of our framework.

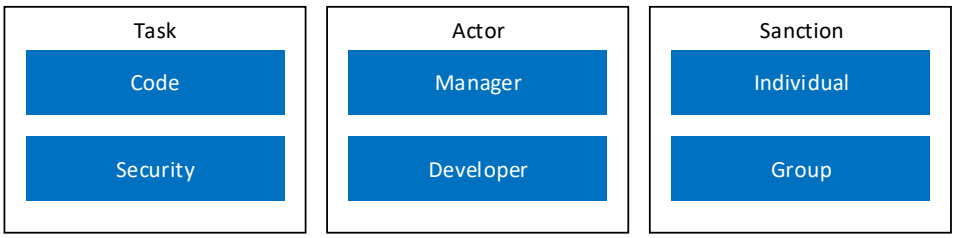

Figure 1: Conceptual Model of Security Practices Adoption Framework

Definition 1. *The security practices adoption framework* $O =< L, A, P, M, E >$ *contains five entities:* L *is the lab, or an organization, where agents reside in.* A *is a set of developer agents who perform tasks pertaining to a time-critical project. Each task corresponds to an artifact of a product* P*.* M *is a manager agent who monitors the coding and security practices of the agents and sanctions agents based on the functionality and security of the product.* E *is everything outside of the lab.*

The following sections describe the attributes and actions of the manager and developer agents.

2.1. Manager

Manager agent is in charge of assigning tasks and sanctions.

2.1.1. Assign Tasks

Modern software development is mostly based on an incremental and iterative approach in which software is developed iteratively, or through repeated cycles, and in small sections at a time [45]. We assume each part of a software corresponds to a task. A task in our model can be in one of the three states: $NotCoded$, $Coded$, $Tested$, as shown in Figure 2. $NotCoded$ indicates the task has not yet been adequately coded, $Coded$ represents an

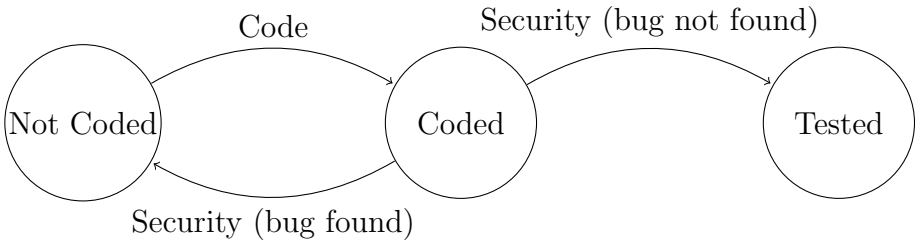

Figure 2: Task States

agent performed *Code* action on the task but not tested for bugs, and *Tested* indicates that an agent performed *Security* action, i.e., running security tools on the task and no bugs were found. A task has the following attributes:

Minimum skill required to code. Each task has an associated minimum skill required to code. If a developer's coding skill is higher than that required skill, it can perform *Code* action on that task. A developer can perform the action *LearnCode* to increase its skill of coding.

Minimum skill required to run security tools. Each task has an associated minimum skill required to perform *Security* action. If a developer's security skill is higher than that required skill, it can perform *Security* action on that task. A developer can perform *LearnSecurity* action to increase its skill of using security tools.

Time required to code. Each task has an associated time required to code. The time is inversely proportional to the coding skill of a developer.

Time required to run security tools. Each task has an associated time required to run security tools. The time is inversely proportional to the security skill of a developer.

To reduce the dimensionality of the complex problem, tasks are assumed to be atomic. Each task corresponds to an artifact. An artifact, Art_j, corresponding to task T_j, has two attributes—functionality (indicates how functional the artifact is) and security (indicates how secure the artifact is). In our model, functionality of an artifact is computed as a function of the coding skill of the corresponding developer agent. Security of the artifact is a function of the skill of using security tools of the developer agent, whether or not *Code* and *Security* action is performed, and whether bugs are found in the artifact.

2.1.2. Assign Sanctions

Sanctions can be individual or group sanctions. In individual sanctioning, developers are subjected to sanctions individually, if they fail to deliver artifacts with the standard set by manager. In a group sanction, the manager only monitors the overall product and sanctions every developer if the product does not meet the standard. Sanctions are applied in the following two cases:

For functionality. The manager monitors functionality of the artifacts periodically and imposes sanctions to (1) individual developer agents who produce artifacts with less functionality than the threshold standards set by the manager (individual sanction) or (2) all developers if the overall product functionality is less than the project threshold standard (group sanction).

For security. Similar to functionality, the manager monitors security of the artifacts periodically and imposes individual or group sanctions.

For both. The manager monitors functionality and security both- of the artifacts periodically, and imposes individual or group sanctions for not meeting any one of the standards.

2.2. Developer Agents

Each developer agent has four attributes:

Tasks. Task are works assigned to a developer agent by the manager agent. The duration of time available to a developer agent to complete the assigned tasks is the deadline. Here, "complete" means both "Code" action and "Security" action are performed on a task. The deadline also marks the end of a project. There are multiple projects in each simulation.

Coding and Security skills. Each developer has a skill of coding and using security tools. The time taken to code or run security tools decreases with an increase in the corresponding skill, and developers can increase their skills by performing *LearnCode* or *LearnSecurity* actions.

Developer health. It represents a developer agent's health and varies according to the completion of the tasks assigned to the agent. The initial value is 100, which is also the maximum health value. An agent's health is subject to change in the model for the following reason: if an agent is unable to meet the functionality and security standards within the deadline, health is reduced proportionally. A threshold of the health is set, below which an agent is considered "dead" and needs to leave the lab. If an agent leaves, it is no longer be able to perform any action and will never return to the simulation [22].

Preference. It is the probability that an agent will choose to do an action [22]. For example, if an agent's preference is 40% for coding, 20% for using security tools, 20% for learning coding and 20% for learning security tools, it has a 40% chance of considering the *Code* action and 20% chance of considering the *Security*, *LearnCode*, and *LearnSecurity* actions. The sum of the preferences is unity.

Definition 2. *A developer agent can choose to perform one of the following actions at each time step. Actions* $= \{Code, Security, LearnCode, LearnSecurity, DoNothing\}$. *Actions are described in detail below.*

*Code***.** A task changes from *NotCoded* to *Coded* state when the assigned developer agent performs *Code* action on the task.

*Security***.** When a developer agent performs *Security* action on a task, the task state changes from *Coded* to *Tested* if no bugs are found. If there are bugs, the state changes from *Coded* to *NotCoded*.

*LearnCode***.** The action *LearnCode* increases the skill of coding of an agent. It results in a decrease in the time to perform *Code* action on a task and increases the functionality of the corresponding artifact.

*LearnSecurity***.** The action *LearnSecurity* increases the skill of using security tools. It results in decrease in time to perform *Security* action on a task and increases the security of the corresponding artifact.

*DoNothing***.** A rational agent may *DoNothing* if there is any reward for doing nothing or all the tasks in a project are *Coded* and *Tested*.

3. Simulation Experiments

The following sections describe characteristics of the simulation experiments.

3.1. Assumptions and Simulation Settings

The following section discusses several assumptions regarding our experiment. We understand these variables are not arbitrary and we do not trivialize the significance of these variables. Further research and data collection should be conducted to replace the assumed variables with refined values. However, as the nature of our simulation is exploratory, we have intuitively assigned values to these variables to demonstrate the dynamics of the framework. These assumptions and other related parameters are as follows:

- The manager has full observability of the system. A developer agent has observability only about tasks assigned to it and its own attributes.
- Developers perform tasks in the order they are assigned. A developer considers coding a task only if its prior task is *Coded*.
- The skill of coding and the skill of running security tasks of a developer are independent. Each skill varies in the range of 0 to 100.
- The expected reward of running security tools on a coded task is higher than the expected reward of coding a task.
- The manager only assigns negative sanctions based on functionality and security compliance.

3.2. Runtime Actions

The total tasks are distributed equally among the active developers. At each tick, the following actions occur:

1. Each developer identifies the available actions to perform at that time tick.
    - Action *Code* is available if there is any task in the *NotCoded* state and the corresponding developer's skill of coding is equal to or higher than the required skill for the task.

Algorithm 1: Developer's decision and task completion

1: **Function** Π (developers, tasks, $f_{decision}$, $f_{changestate}$)
2: decisions $\leftarrow$ $f_{decision}$ (developers, tasks);
3: **for all** developers in decisions **do**
4: **if** decision is to *Learn* **then**
5: skills$\leftarrow$ $f_{upskills}$ (skills, δskill);
6: **else if** decision is to *Code* **then**
7: tasks $\leftarrow$ $f_{changestate}$ (tasks);
8: **else if** decision is to *Security* **then**
9: tasks $\leftarrow$ $f_{changestate}$ (tasks, bugs);

- Action *Security* is available if there is any task in *Coded* state and the corresponding developer's skill of running security tool is equal or higher than the required security skill for the task.
- Action *LearnCode* is available if the corresponding developer's coding skill is less than 100.
- Action *LearnSecurity* is available if the corresponding developer's skill of running security tool is less than 100.

2. Each developer compares the expected reward of each action as follows:

Expected reward of *Code* action R_{Code}. It is the product of units of task that can be completed with current coding skill within time remaining (N_{code_task}), and the reward for coding a unit task (R_{code_task}).

$$R_{Code} = N_{code\_task} * R_{code\_task}$$

Expected reward of *Security* action $R_{Security}$. It is the product of units of task that can be tested with the current security skill within the time remaining (N_{test_task}), and the reward for testing a unit task (R_{test_task}).

$$R_{Security} = N_{test\_task} * R_{test\_task}$$

Expected reward of *LearnCode* action $R_{LearnCode}$. It is the product of the units of task that can be coded if the current time step is spent on learning coding ($N_{code\_task\_if\_learn}$), i.e., units of tasks that can be coded with an increased skill within (time remaining–1) time steps, and the reward for coding unit task.

$$R_{LearnCode} = N_{code\_task\_if\_learn} * R_{code\_task}$$

Expected reward of *LearnSecurity* action $R_{LearnSecurity}$. It is the product of the units of task that can be tested if the current time step is spent on learning security tools ($N_{test\_task\_if\_learn}$), i.e., units of tasks that can be tested with an increased skill within (time remaining -1) time steps, and the reward for testing a unit task.

$$R_{LearnSecurity} = N_{test\_task\_if\_learn} * R_{test\_task}$$

3. Each developer identifies its action. The action with the highest product of preference and expected reward is identified by the developer (function $f_{decision}$ in Algorithm 1). The state of each developer is updated (function $f_{changestate}$) as follows:
   - If *Code* action is performed, a developer remains *busy* for time steps equal to the time required for coding that task. After that, the state of the task changes from $NotCoded$ to $Coded$. Once a task state changes from $NotCoded$ to $Coded$, the functionality of the corresponding artifact is generated as a random number in the range of [skill of coding, 100] and the security of the artifact in the range of [skill of security, 100].
   - If *Security* action is performed, a developer remains *busy* for time steps equal to the time required for testing that task. After that, the state of the task changes from $Coded$ to $Tested$ if no bug is found and to $NotCoded$ state if any bug is found. The probability of finding a bug is a random number in the range [0, (100-security skill)]. Once a task state changes from $Coded$ to $Tested$, the security of the corresponding artifact is increased by a percentage of the skill of security of the corresponding developer.
   - If *Learn Code* or *Learn Security* action is performed, the corresponding skill of the developer increases.
4. When the time reaches deadline, a project ends and the manager performs sanctions based on compliance of functionality and security requirements (Algorithm 2). When a developer is sanctioned for functionality, its preference for $Code$ action and $LearnCode$ action increases, and when a developer is sanctioned for security, its preference for $Security$ action and $LearnSecurity$ action increases. A new project begins at the end of a project.

Algorithm 2: Manager's sanction

1: **Function** ϑ (developers, Artifacts, threshold, $type_{sanction}$, $f_{sanction}$)
2: violations $\leftarrow f_{sanction}$(artifacts, threshold, Stype);
3: **for all** artifacts in violations **do**
4: **if** developer is sanctioned **then**
5: preferences $\leftarrow f_{uppref}$(preferences, $type_{sanction}$);

3.3. Scenarios and Metrics

First, we compare the performances of three different types of developers. The first type of developer agent always selects *Code* if coding is an option. They perform *LearnCode* action only if learning is required. Once all the tasks are coded, they perform *Security* action and *LearnSecurity* action (only if more learning is required to reach threshold skill). The second type of developer agent codes a task and runs security tests of the code immediately when the coding of a task is complete. They learn coding or running security tools as required. These developer agents start coding the next task when the previous task is coded and tested. The third type of developer agent always learns first. These agents learn coding to reach the maximum coding skill and then learn security to reach the maximum security skills. Once their skills are at maximum levels, they code and run security tests. We compare the sanction, sanction efficacy and overall group health for these three types of developer agents. Because they have predefined strategies, these developer agents do not learn or change actions in response to sanctions.

Next we consider a group of developers who learn and update their preferences in response to sanctions. We compared three additional group of developers. The first group of developers have no preference, i.e., the preferences for *Code*, *Security*, *LearnCode* and *LearnSecurity* actions are equal (equal to 25). The second group of developers have higher preference for coding, that is, the preferences for *Code* action (85) is higher than the actions *Security*, *LearnCode* and *LearnSecurity* (each set at 05). The third group of developers have higher preference for running security tools, that is, the preferences for *Security* action (85) is higher than the actions *Code*, *LearnCode* and *LearnSecurity* (05). The following four metrics are measured over the course of the simulation:

Tasks tested %. It is the percentage of tasks in a project in which a developer performed *Security* action and no bugs are found. In other

words, tasks tested (%) is the ratio of tasks in *Tested* state and total tasks, measured at the deadline.

Time spent on security tasks %. It is the total time steps spent by all developers on *Security* and *LearnSecurity* actions as a percentage of the total time steps of the project.

Sanctions %. It is the number of times developers are sanctioned by the manager as a percentage of maximum number of sanctions possible, in the simulation.

Sanction efficacy. It is computed as the ratio of the number of developers that are sanctioned once and the total number of developers in the simulation. Sanction efficacy implies that the sanctioning changed the preference of the developer so that it was never sanctioned again.

3.4. Simulation of Bug Reports

To make the simulation more realistic, we incorporated real data. Specifically, we generate a time history of bugs using the bug report data for a developing software product— Eclipse Java Development Tools [46]. The report contains bug ids, reported time, summaries, and commits. Because Eclipse releases new versions around the end of June every year, we consider a time frame of one year between two releases, which corresponds to a project in our simulation. We calculated the number of bugs reported each month and normalized it:

$$normalized\_data = \frac{number\_of\_bugs}{maximum\_number\_of\_its\_release} * 100.$$

The normalization is to relieve the side effect of fewer bugs in later versions. We have 149 records of normalized data from October 2001 to February 2014, which are used to generate the data we used in the simulation. We used the following formula to generate simulation data from the normalized data:

$$simulation\_data = normalized\_data * \frac{random(80, 120)}{100}.$$

For $simulation\_data > 100$, we consider it as 100. We generated 50 batches of simulation data according to the normalized data to be used in 50 simulations. Each simulation contains 10 project cycles, where each cycle in equivalent to one year of records. Each project cycle is assumed to run for 60 time steps. The experiment parameters are described in Table 1.

Table 1: Experiment parameters

Parameter	Value
No of simulations	50
No of projects	10
No. of Developers	100
Tasks per project	500
Project duration	60
Preference for coding for coding preferred	85
Preference for other action for coding preferred	05
Preference for testing for testing preferred	85
Preference for other action for testing preferred	05
Preference for all actions for no preferences	25
Maximum skill	100
Variables with normal distribution	μ (σ)
Time required to code a task	6 (1)
Time required to test a task	5 (1)
Skill of developers (initialization)	50 (5)
Skill required for a task (initialization)	50 (5)
Health of developers (initialization)	95 (5)

4. Evaluation

4.1. Emergence in Developers' Responses

The security adoption is not dominated by the skills of developers; rather it emerges through the complex and dynamic interactions among the skills of developers, their preferences, task properties, and task assignments. Security actions and tasks tested vary across scenarios. Developers with preferences and developers with fixed strategy are similar in actions and tasks tested for the no sanction scenarios. Because of the differences in adaptive responses, their responses vary under sanctions as demonstrated in Figure 3-6. Results demonstrated through these figures are further described in the following sections.

4.2. Fixed vs Adaptive Developers

We compare the performances of developers with different fixed strategies and preferences. Figure 3a and 3b show the mean percentage of tasks tested

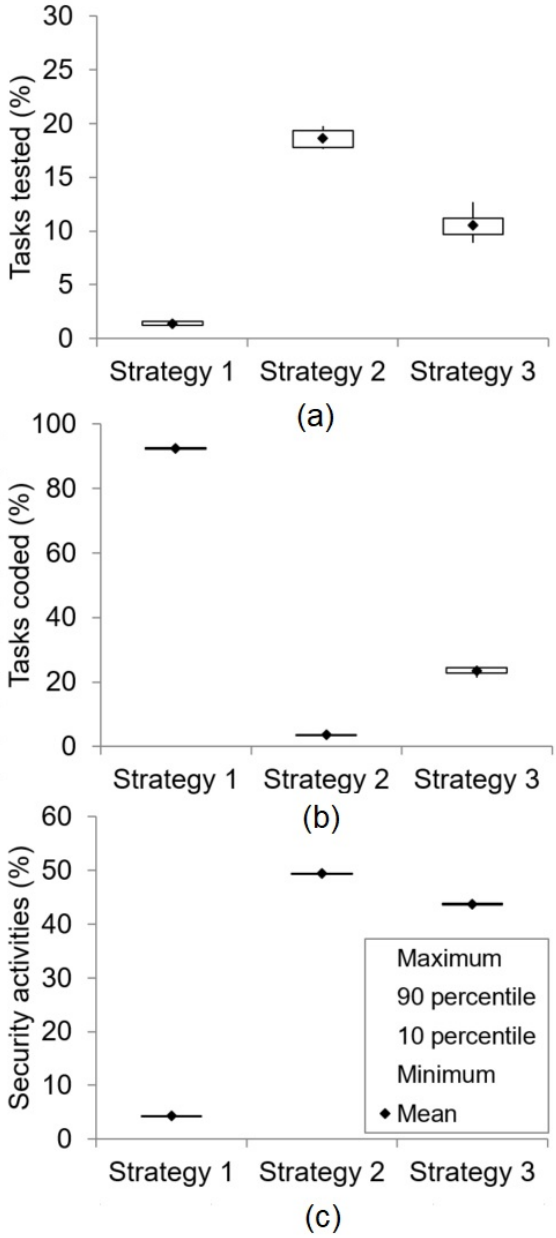

Figure 3: Tasks (a) coded (b) tested and (c) security activities for no sanction scenario of fixed strategy developers (Strategy 01 is Code⇔LearnCode→Security⇔LearnSecurity, Strategy 02 is Security⇔LearnSecurity→Code⇔LearnCode, Strategy 03 is LearnCode→LearnSecurity→Code→Security)

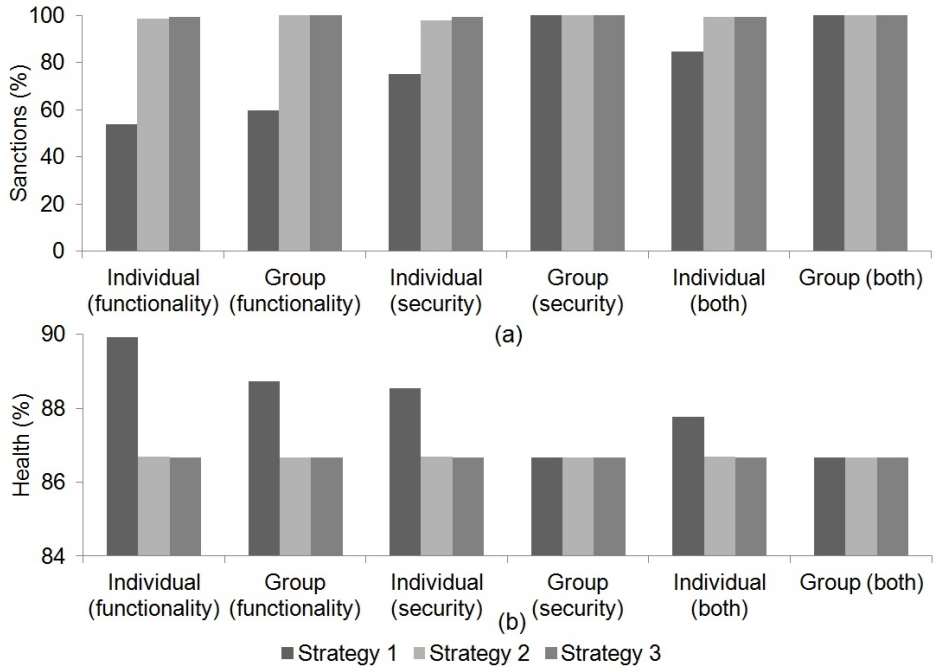

Figure 4: (a) Sanctions, and (b) Health for no sanction scenario of fixed strategy developers (Strategy 01 is Code⇔LearnCode→Security⇔LearnSecurity, Strategy 02 is Security⇔LearnSecurity→Code⇔LearnCode, Strategy 03 is LearnCode→LearnSecurity→Code→Security)

and coded, respectively, by a developer in a project, and Figure 3c shows the corresponding security activities measured as percentage of *Security* and *LearnSecurity* actions performed in each timestep. The mean is calculated for 50 simulations. Among the fixed strategies, developers who perform *Security* or *LearnSecurity* actions first (strategy 2) and then perform *Code* and *LearnCode* actions have the highest mean of percentage of tasks tested (19%). This group of developers also has the highest mean of percentage (50%) of security activities among the three strategies compared. The group of developers who perform *Code* or *LearnCode* first, and then perform *Security* or *LearnSecurity* actions has the highest mean percentage (93%) of tasks coded but also the lowest percentage (1.4%) of tasks tested. This group also spends the least amount of time on security activities (4%), as shown in Figure 3.

Among the adaptive developers, or developers that update their preferences when sanctioned (Figure 5), developers with no preferences or preferences for testing have a higher mean of percentage of tasks tested (10.9% and 11.1%) and security activities (28%), compared to developers with preference for coding (1.4% tasks tested and 4.4% security activities). As shown in Figure 3b, the 10 and 90 percentile values are very close to the mean, which indicates that most of the simulations produced percentage of tasks tested close to the mean. The same is true for most of the simulations for adaptive developers. The highest variations are observed in the case of group sanctions for security and both scenarios.

Table 2: Changes in tasks tested (%) and security activities (%) with change in preference over 50 simulations (Tasks coded in no preference, τ_1=9.8–12.2 and security activities in no preference, μ_1=27.5–28.1)

Preference	Tasks tested (τ_2)	$\tau_1 > \tau_2$	$\tau_1 < \tau_2$	$\tau_1 \neq \tau_2$ at 95% CI
For coding	1.0–1.8	100%	0%	100%
For testing	9.8–12.7	34%	66%	2%
Preference	Security activities (μ_2)	$\mu_1 > \mu_2$	$\mu_1 < \mu_2$	$\mu_1 \neq \mu_2$ at 95% CI
For coding	4.3–4.5	100%	0%	100%
For testing	27.7–28.4	4%	96%	0

4.3. Preferences

Among the three developer groups with different preferences, developers with preferences for testing have the highest security practices (27.7-28.4%) and tasks tested (9.8 to 12.7%). Developers with preferences for coding have the lowest security practices and tasks tested. Table 2 compares the tasks tested and security activities across different preferences. As shown, developers with preference for coding tested less than the developers with no preference in 100% of the simulations. Developers with preference for testing tested more than the developers with no preferences in 66% of the simulations. Similar observations can be made in security activities where developers with no preference had higher security activities than developers with no preferences in 100% of simulations and lower than the developers with preference for testing in 96% of simulations. A two-tailed student's t-test assuming unequal variance shows 100% of the simulations have statistical significance for rejecting the null hypothesis that the means of tasks tested and security activities are equal to that of developers of coding, whereas only 2% of the simulations can show the same for developers with preference for testing in case of tasks tested.

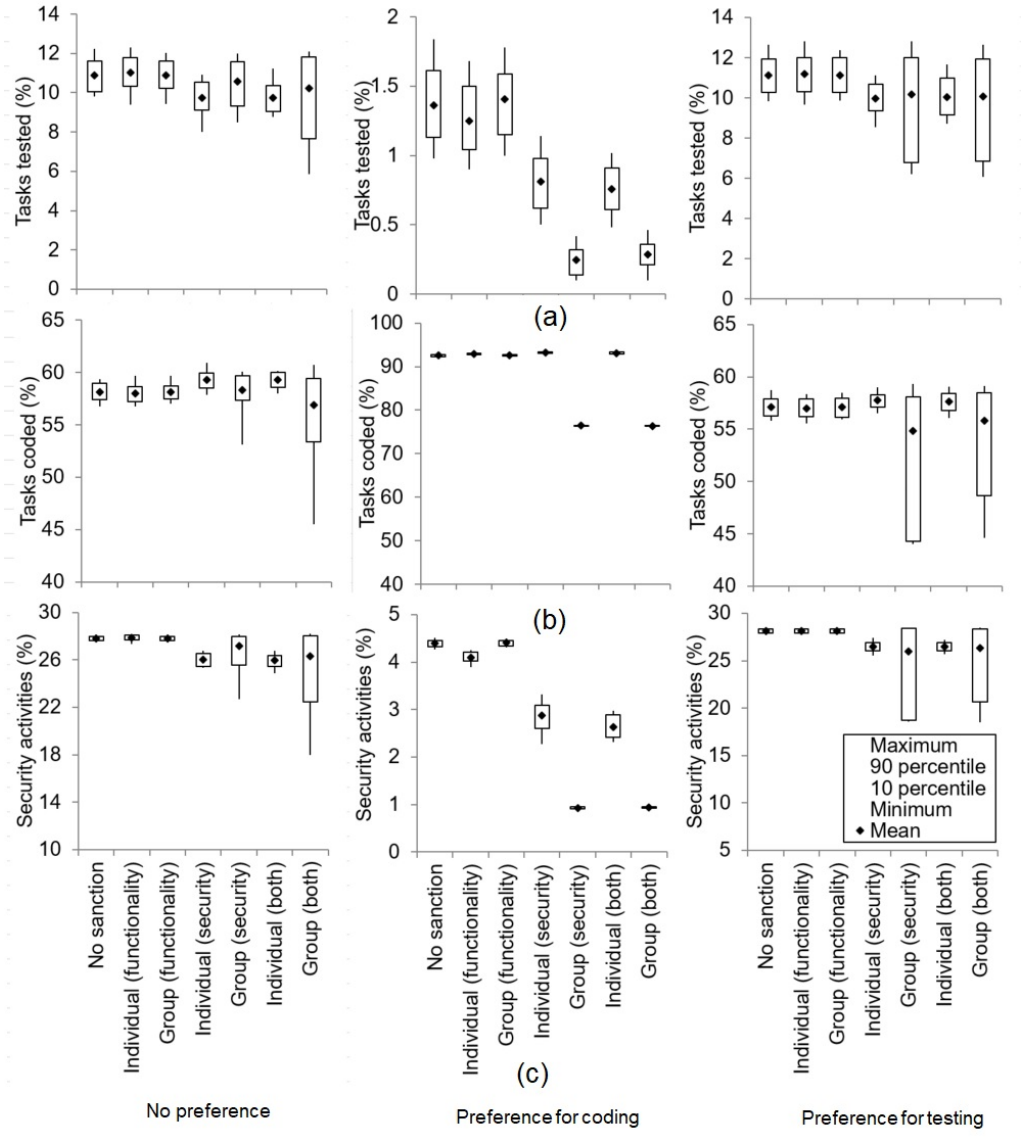

Figure 5: (a) Tasks tested, (b) Tasks coded and (c) security activities for sanction scenarios of no preference, preference for coding and preference for testing

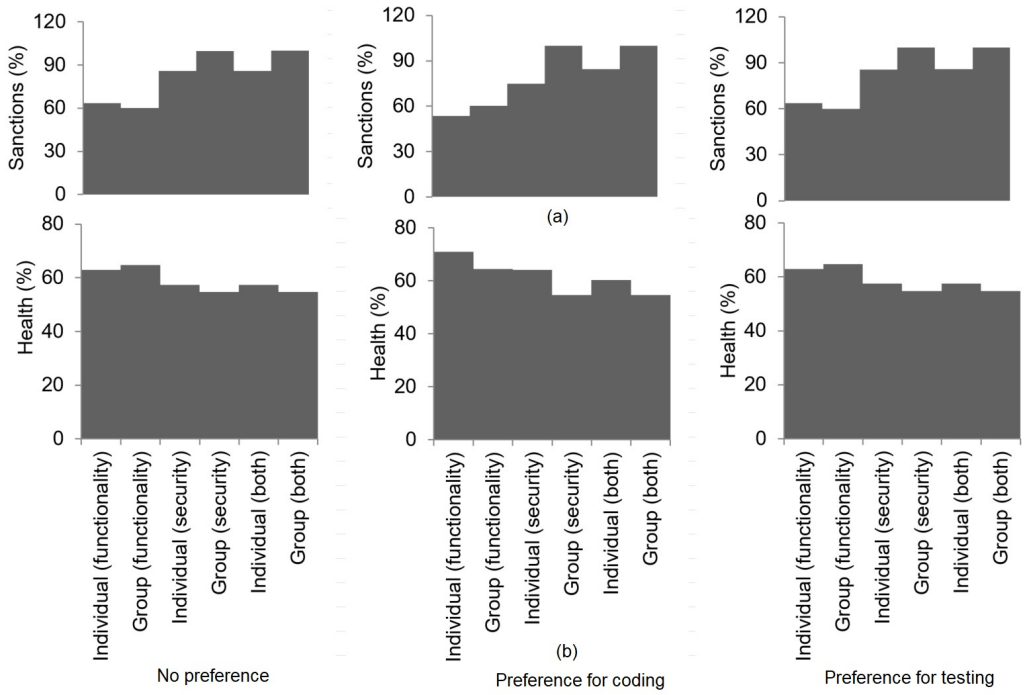

Figure 6: (a) Sanctions, and (b) Health for for sanction scenarios of no preference, preference for coding and preference for testing

Table 3: Changes in tasks tested (%) with sanctions over 50 simulations (Tasks tested in no sanction (τ_1) for no preference ranges between 9.8–12.2, for preference for coding between 1.0–1.8 , and for preference for testing between 9.8–12.7)

Preference	Sanction	Tasks tested (τ_2)	$\tau_1 > \tau_2$	$\tau_1 < \tau_2$	$\tau_1 \neq \tau_2$ at 95% CI
No preference	Individual (functionality)	9.38-12.3	44%	56%	0%
	Group (functionality)	9.4–12.0	46%	54%	4%
	Individual (security)	8.0–10.9	100%	0%	34%
	Group (security)	8.5–12.0	52%	48%	12%
	Individual (both)	8.8–11.2	96%	2%	32%
	Group (both)	6.1–12.8	62%	38%	30%
Preference for coding	Individual (functionality)	0.9-1.7	70%	24%	0%
	Group (functionality)	1–1.8	34%	62%	0%
	Individual (security)	0.5–1.1	100%	0%	62%
	Group (security)	0.1–0.4	100%	0%	100%
	Individual (both)	0.5–1.0	100%	0%	74%
	Group (both)	6.1–12.8	100%	0%	100%
Preference for testing	Individual (functionality)	9.7-12.8	46%	54%	2%
	Group (functionality)	9.9–12.4	46%	54%	0%
	Individual (security)	8.5–11.1	94%	6%	34%
	Group (security)	6.2–12.2	58%	42%	32%
	Individual (both)	8.7–11.7	100%	0%	24%
	Group (both)	6.1–12.7	70%	30%	28%

4.4. Sanctions

Figure 4 demonstrates the sanctions and corresponding health under fixed strategies. As shown in 4a, developers with fixed strategy 1, i.e., developers who code or learn code first, are sanctioned least. This group of developers also has the highest percentage of tasks coded. As a task is coded, functionality and security values are assigned to the corresponding artifacts (section 3.2. When the developer performs *Security* action on the same task, the security value is further increased. However, since the developers in strategy 1 coded significantly higher percentage of tasks than the other two strategies, they ended up less sanctioned. The number of tasks coded (and the corresponding security value of the artifact) dominated in this case over the increase of security by performing *Security* action. This group of developers have the highest health as shown in 4c. The developers were sanctioned most for group sanction for security.

Figure 6 shows the sanctions and health of adaptive developers. Similar to fixed strategies, in this case, the sanctions are also guided by number of tasks coded. The results highlight the role of the time-constraints for developers in adoption of security practices. As developers with no preference or preference for security tested more (10%) tasks but ended with higher sanctions since they coded less (35%). As shown in figure 6a, across all preferences, developers are sanctioned highest under the group sanction for functionality and the group sanction for functionality and security. The mean of developers' health is also lowest for these sanctions. The other sanction scenarios are comparable. Though it is counter-intuitive, the highest sanction does not yield the highest task tested, shown in Table 3. The repetitive individual sanction reduces the health of a group of developers, eventually leading to lower retention of developers and thereby lowering the mean percentage of tasks tested. The sanctioning for functionality increased the number of tasks tested for developers with all preferences. The sanctioning for security also increased the tasks tested but lower retention of developers through repetitive sanctions lowered the overall mean.

Sanction efficacy decreases with increase in sanctions. For the scenarios presented here, sanction efficacy is zero for most sanction cases. Individual sanctioning for functionality (12.2%) and security (1.5%) have the only efficacy among the adaptive developers who have preferences for coding.

4.5. Threats to validity

Threats mitigated. We identified and mitigated the three threats.

1. Difference in skills. In reality, developers have different skills thus simulating with same or equal skills introduces a threat of skill difference. To mitigate the threat of skill difference, we seed the simulation with developers with different skills.
2. Difference in developer strategies. Developers working on real projects do not work in the same way to complete the assigned tasks. Thus simulating with only one strategy introduces the threat of difference in developer's strategy. To mitigate this threat, we compare developers with various fixed and adaptive strategies under different sanction mechanisms.
3. Reliability of the data. Since we simulate the task assignment and probability of bugs, a threat is whether the data or the value of the variables seeded in the simulation is reliable. To mitigate this threat of reliability of the data, we seed the simulation based on real data of Eclipse Java Development Tools bug reports spanned across a period of 13 years [46].

The research conducted here relies on a number of assumptions. First, we assumed that all the developers in the organization use only one security analysis tool. Thereby, by learning they can improve their skill of using that particular tool. The improvement in skill for different developers in the organization due to learning was also assumed to be the same. In reality, different developers may prefer using different security analysis tools and the improvements in skill might be different for spending the same amount of time in learning. Second, we assumed that a developer can only perform one action at a time tick. In reality, developers may accomplish multiple tasks simultaneously, for example, they can run a security tool, and simultaneously read a tutorial till the security tool completes its execution. Third, we assumed monotonic functions for changes in health, preferences, and skills of developers in response to sanctions. The changes may vary in time and in person to better represent reality. Fourth, we assumed only negative sanctions in this study. Positive sanctions may influence preferences and health of a developer, as well. In general, evaluating software quality is a complex process [47, 48, 49] that may have time delay and may impact the vulnerability of the code. To reduce dimensionality of the analysis, we assumed that the manager receives evaluation of functionality and security performances of the product immediately after product release and can thereby assign sanctions toward improving those for the next project.

The kinds of knowledge that agent systems rely on require specialized machinery, both for knowledge mining and by way of instrumentation for data collection [50]. We propose a framework for modeling MAS for sanction-based adoption of security practices here. Collecting real-life data to better inform the modeling may include developer surveys and analysis of time sheets that explain distribution of time spent by developers in coding, running security tools, learning and other work. Surveys can also be designed to assess developers' preferences, and interviews can be conducted with managers to characterize sanction mechanisms that are used to mitigate the quality and timeliness of project deliverables.

5. Discussion

Our framework aims to simulate the adoption of security practices among developers toward assessing sanctioning mechanisms. We have presented a novel approach using concepts of decision theory to model the adoption of security practices. We have further demonstrated the application of the framework by using real world data for bug reports from for Eclipse Java Development Tools [46]. The study compares the security practices under different sanctioning mechanism at different levels by comparing tasks that are tested using security tools, time spent on security practices, and overall changes under sanctions. It also assists in comparing functionality and security compliance in different projects for different sanctions by analyzing the recurrence of norm violation after sanctioning and monitoring developers' health under different sanctioning mechanism. Our exploration of system level performance through variable sanction type has yielded some interesting conjectures. For example, we observed that security practices of developers are significantly dictated by preferences of developers. We also observed that repetitive sanctioning may yield lower retention of developers and reduce overall security practices.

Previous studies highlight the role of social factors in adoption of security practices. A survey-based study observed that inquisitiveness may play a strong role in adoption of security practices[2]. The same study also observed that the perceived importance of security may not influence adoption as much as other factors[2]. Another survey-based study [10] observed that users working on web applications have different priorities from those working on desktop applications and that the adoption may vary over time while it is under development to near release. In our framework, we have associated

the adoptions of security practices to preference of developers. The preference may vary across developers and over time as the perceived utility of security practices change. Using real-world data we have demonstrated with statistical significance that if developers have preference for coding than that for learning or running security tests, they may end performing less security actions and testing less tasks for bugs.

Organizational factors may also play an important role in the adoption of security practices [51]. Each organization may have policies and standard pertaining to the security practices. Previous studies observed that users' willingness to review warnings and fix issues depends on project characteristics and organization [10]. We demonstrate the role of organizational interference in promoting security practices through the inclusion of a manager agent and the sanction mechanisms. The developers in the framework may have different skill-level and the performance of the artifacts depend on the corresponding developer. When the performance of the overall product is evaluated, the developers may be subjected to (group) sanction. The sanction may change the preference and the health of the developer which impacts the selection of actions in the following project and the coding and security performance of the groups. We observed the sanction mechanism, standards and frequencies of the manager agent has impacts on developers' preference, actions and overall security performance of the product. There are many discussions in literature that looks into the role of sanctions in shaping moral judgments and compliance norms [30, 52]. Our study observes that repetitive sanctions may eventually reduce adoption of security practices in cases where the health falls below certain threshold.

The framework can be applied to identify the best management intervention techniques to improve security practices. Many organizations keep a formal record of developer activities in the form of a time sheet which can be utilized to track developer activities through out a project cycle. The skill and preference of developers can be assessed based on performance and previous choices of actions in similar projects. Survey and interviews of management professionals may also be useful in identifying the impacts of sanctions. One interesting extension of this study could be to look into decentralized optimizations of security practices to maximize the security performances of the end product. Incorporation of multidisciplinary concepts from decision theories and measuring resilience and liveliness of the system in connection to the sanctions, among others, will help to get a holistic view of the system to guide effective sanctions toward security practices.

Acknowledgment

We thank Jon Doyle for help and guidance in developing the model. We thank Laurie Williams, William Enck, Özgür Kafalı, David Wright, Christopher Theisen, Sarah Elder, Lena Leonchuk, Lindsey McGowen and the Science of Security lablet for the feedback. We thank the US Department of Defense for support through the Science of Security Lablet grant to North Carolina State University and the anonymous reviewers for helpful comments.

[1] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: Proceedings of the 35th International Conference on Software Engineering, IEEE Press, San Francisco, 2013, pp. 672–681.

[2] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, T. Zimmermann, Quantifying developers' adoption of security tools, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE), ACM, Bergamo, Italy, 2015, pp. 260–271.

[3] Klocwork: Source code analysis tool, online; accessed 30-May-2017 (2017).
URL `https://www.klocwork.com/products-services/klocwork`

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A few billion lines of code later: Using static analysis to find bugs in the real world, Communications of the ACM 53 (2) (2010) 66–75.

[5] K. Kina, M. Tsunoda, H. Hata, H. Tamada, H. Igaki, Analyzing the decision criteria of software developers based on prospect theory, in: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, Osaka, Japan, 2016, pp. 644–648.

[6] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM Sigplan Notices 39 (12) (2004) 92–106.

[7] N. Nethercote, J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, in: ACM Sigplan notices, Vol. 42, ACM, 2007, pp. 89–100.

[8] NetBeans IDE, online; accessed 30-May-2017 (2017).
URL http://www.netbeans.org

[9] Eclipse, online; accessed 30-May-2017 (2017).
URL http://www.eclipse.org/

[10] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, IEEE software 25 (5).

[11] C. W. Araújo, I. Nunes, D. Nunes, On the effectiveness of bug predictors with procedural systems: A quantitative study, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2017, pp. 78–95.

[12] V. Venkatesh, F. D. Davis, A theoretical extension of the technology acceptance model: Four longitudinal field studies, Management Science 46 (2) (2000) 186–204.

[13] P. Legris, J. Ingham, P. Collerette, Why do people use information technology? a critical review of the technology acceptance model, Information & management 40 (3) (2003) 191–204.

[14] R. W. Zmud, Diffusion of modern software practices: Influence of centralization and formalization, Management Science 28 (12) (1982) 1421–1431.

[15] E. M. Rogers, Diffusion of innovations, Simon and Schuster, 2010.

[16] G. Madey, V. Freeh, R. Tynan, The open source software development phenomenon: An analysis based on social network theory, AMCIS 2002 Proceedings (2002) 247.

[17] P. J. Carrington, J. Scott, S. Wasserman, Models and methods in social network analysis, Vol. 28, Cambridge university press, 2005.

[18] R. W. Selby, A. A. Porter, Learning from examples: Generation and evaluation of decision trees for software resource analysis, IEEE Transactions on Software Engineering (TSE) 14 (12) (1988) 1743–1757.

[19] G. Büyüközkan, D. Ruan, Evaluation of software development projects using a fuzzy multi-criteria decision approach, Mathematics and Computers in Simulation 77 (5) (2008) 464–475.

[20] J. O. Berger, Statistical decision theory and Bayesian analysis, Springer Science & Business Media, 2013.

[21] V. Dignum, F. Dignum, Contextualized planning using social practices, in: International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems, Springer, 2014, pp. 36–52.

[22] H. Du, B. Narron, N. Ajmeri, E. Berglund, J. Doyle, M. P. Singh, Understanding sanction under variable observability in a secure, collaborative environment, in: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security (HotSoS), ACM, Urbana-Champaign, 2015, pp. 12:1–12:10.

[23] M. P. Singh, Norms as a basis for governing sociotechnical systems, ACM Transactions on Intelligent Systems and Technology (TIST) 5 (1) (2013) 21:1–21:23.

[24] M. Mashayekhi, H. Du, G. F. List, M. P. Singh, Silk: A simulation study of regulating open normative multiagent systems, in: Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI), AAAO Press, New York, 2016, pp. 373–379.

[25] J. Morales, M. López-Sánchez, J. A. Rodríguez-Aguilar, M. Wooldridge, W. W. Vasconcelos, Minimality and simplicity in the on-line automated synthesis of normative systems, in: Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, Paris, 2014, pp. 109–116.

[26] J. Morales, M. López-Sánchez, J. A. Rodríguez-Aguilar, M. Wooldridge, W. W. Vasconcelos, Synthesising liberal normative systems, in: Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, Istanbul, 2015, pp. 433–441.

[27] D. Avery, H. K. Dam, B. T. R. Savarimuthu, A. Ghose, Externalization of software behavior by the mining of norms, in: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, 2016, pp. 223–234.

[28] T. Keller, B. T. R. Savarimuthu, Facilitating enhanced decision support using a social norms approach, Journal of Electronic Commerce in Organizations (JECO) 15 (2) (2017) 1–15.

[29] P. Noriega, A. K. Chopra, N. Fornara, H. L. Cardoso, M. P. Singh, Regulated MAS: Social Perspective, in: G. Andrighetto, G. Governatori, P. Noriega, L. W. N. van der Torre (Eds.), Normative Multi-Agent Systems, Vol. 4 of Dagstuhl Follow-Ups, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013, pp. 93–133.

[30] D. D. Heckathorn, Collective sanctions and compliance norms: A formal theory of group-mediated social control, American Sociological Review 55 (3) (1990) 366–384.

[31] L. G. Nardin, T. Balke-Visser, N. Ajmeri, A. K. Kalia, J. S. Sichman, M. P. Singh, Classifying sanctions and designing a conceptual sanctioning process model for socio-technical systems, The Knowledge Engineering Review (KER) 31 (2016) 142–166.

[32] M. Xenitidou, B. Edmonds, The complexity of social norms, Springer, 2014.

[33] B. T. R. Savarimuthu, S. Cranefield, Norm creation, spreading and emergence: A survey of simulation models of norms in multi-agent systems, Multiagent and Grid Systems 7 (1) (2011) 21–54.

[34] B. T. R. Savarimuthu, S. Cranefield, M. K. Purvis, M. A. Purvis, Norm emergence in agent societies formed by dynamically changing networks, Web Intelligence and Agent Systems: An International Journal 7 (3) (2009) 223–232.

[35] B. Savarimuthu, M. Purvis, M. Purvis, S. Cranefield, Social norm emergence in virtual agent societies, Declarative Agent Languages and Technologies VI (2009) 18–28.

[36] K. H. Dam, M. Winikoff, Comparing agent-oriented methodologies, in: Agent-Oriented Information Systems, Springer Berlin Heidelberg, 2004, pp. 78–93.

[37] H. K. Dam, A. Ghose, An agent-based framework for distributed collaborative model evolution, in: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, ACM, 2011, pp. 121–130.

[38] T. Meyer, A. Ghose, S. Chopra, Multi-agent context-based merging, in: Proceedings of Common Sense, 2001.

[39] A. Ghose, T. B. R. Savarimuthu, Norms as objectives: Revisiting compliance management in multi-agent systems, in: International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems, Springer Berlin Heidelberg, 2012, pp. 105–122.

[40] J. Liu, Z. Wei, Agent-based computation of decomposition games with application in software requirements decomposition, in: Multi-agent and Complex Systems, Springer, 2017, pp. 165–179.

[41] J. Rajamäki, Cyber security, trust-building, and trust-management: As tools for multi-agency cooperation within the functions vital to society, in: Cyber-Physical Security, Springer, 2017, pp. 233–249.

[42] H. Yang, F. Chen, S. Aliyu, Modern software cybernetics: New trends (2017).

[43] J. B. de Vasconcelos, C. Kimble, P. Carreteiro, Á. Rocha, The application of knowledge management to software evolution, International Journal of Information Management 37 (1) (2017) 1499–1506.

[44] G. D'Angelo, S. Ferretti, Lunes: Agent-based simulation of p2p systems, in: High Performance Computing and Simulation (HPCS), 2011 International Conference on, IEEE, 2011, pp. 593–599.

[45] M. Choetkiertikul, H. K. Dam, T. Tran, A. Ghose, J. Grundy, Predicting delivery capability in iterative software development, IEEE Transactions on Software Engineering.

[46] A. N. Lam, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, Combining deep learning with information retrieval to localize buggy files for bug reports (n), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, 2015, pp. 476–481. doi:10.1109/ASE.2015.73.

[47] H.-W. Jung, S.-G. Kim, C.-S. Chung, Measuring software product quality: A survey of iso/iec 9126, IEEE software 21 (5) (2004) 88–92.

[48] E. Van Veenendaal, R. Hendriks, R. Van Vonderen, Measuring software product quality.

[49] B. W. Boehm, J. R. Brown, M. Lipow, Quantitative evaluation of software quality, in: Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, 1976, pp. 592–605.

[50] A. Ghose, Agents in the era of big data: What the "end of theory" might mean for agent systems., in: PRIMA, 2013, pp. 1–4.

[51] S. Xiao, J. Witschey, E. Murphy-Hill, Social influences on secure development tool adoption: why security tools spread, in: Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing, ACM, 2014, pp. 1095–1106.

[52] D. E. Warren, K. Smith-Crowe, Deciding what's right: The role of external sanctions and embarrassment in shaping moral judgments in the workplace, Research in organizational behavior 28 (2008) 81–105.