2012 IEEE Sixth International Conference on Software Security and Reliability

On the Effective Use of Security Test Patterns

Ben Smith and Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
[ben_smith, laurie_williams]@ncsu.edu

Abstract— **Capturing attacker behavior in a security test plan allows the systematic, repeated assessment of a system's defenses against attacks. To address the lack of security experts capable of developing effective black box security test plans, we have empirically developed an initial set of six black box security test patterns. These patterns capture the expertise involved in creating a black box security test plan in the same way that software design patterns capture design expertise. Security test patterns can enable software testers lacking security expertise (in this paper, "novices") to develop a test plan the way experts could.** ***The goal of this paper is to evaluate the ability of novices to effectively generate black box security tests by accessing security expertise contained within security test patterns.*** **We conducted a user study of 47 student novices, who used our six initial patterns to develop black box security test plans for six requirements from a publicly available specification for electronic health records systems. We created an oracle for the security test plan by forming a panel of researchers who manually completed the same task as the novices. We found that novices will generate a similar black box test plan to the oracle when aided by the six black box security test patterns.**

Keywords-security; vulnerability; patterns; testing; black box; user study

I. INTRODUCTION

In 2010, Jim Gosler, a fellow at the Sandia National Laboratory who works on countering attacks on U.S. networks, claimed that there are approximately 1,000 people in the country with the skills needed for cyber defense. Gosler went on to say that 20 to 30 times that many are needed [1]. Additionally, the CEO of Mykonos Software security firm indicated that today's graduates in software engineering are unprepared to enter the workforce because they lack a solid understanding of how to make their applications secure [2]. Particularly due to this shortage of security expertise [3], the development community needs a vehicle to capture and disseminate knowledge about how to assess whether software systems have adequate defenses against malicious users.

Capturing attacker behavior in a security test plan allows the systematic, repeated assessment of a system's defenses against an attack or class of attacks. We adapt the notion of a software design pattern as proposed by Gamma et al. [4] to the domain of black box security testing. A design pattern is a description of a recurring problem and a description of the core solution to the design problem that is described "in such a way that you can use this solution a million times over, without ever doing it the same way twice" [5]. A software *security test* pattern is a recurring security problem, and the description of a test case that reveals that security problem, that is described such that the test case can be instantiated a million times over, without ever doing it the same way twice. Just as design patterns capture design knowledge into a reusable medium [4], software security test patterns capture security-testing knowledge into a reusable medium.

We developed an initial set of six security test patterns using a two-step empirical grounded theory approach [6]. We first produced a black box test case that would successfully expose a vulnerability from the CWE/SANS Top 25[1] using security expertise. The Top 25 is a list of the most dangerous programming errors that can lead to serious vulnerabilities in software. We repeated this process until we covered all of the Top 25 with at least one test case, and then we categorized and grouped the test cases by similarities in their test procedure and approach. Once the organization of the tests was complete, we extracted the similar parts amongst all the groups and we obtained the six intial patterns. We describe these six initial patterns in more detail in Section III. In this paper, we investigate how software testers who have relatively low security training or expertise (henceforth "novices") use these six patterns to develop their own black box security test plans.

Software design patterns have enabled practitioners to access and re-use the design expertise contained within software design patterns to make informed design decisions [7]. Our theory is that our software security test patterns can enable novices to write black box secruity test plans in a similar way to how the experts could.

The goal of this paper is to evaluate the ability of novices to effectively generate black box security tests by accessing security expertise contained within security test patterns. For our patterns to be effective, the patterns and their application must be *accessible*, that is, they must be easy to use and easy to understand. We measured the accessibility of the patterns in three ways: 1) whether the novices used the patterns to generate a similar black box test plan as experts could; 2) the amount of time consumed by the novices as they used the patterns; and 3) the novices' subjective opinions about the usefulness of the patterns after using them to generate black box security test plans. To address this goal, we conducted a

[1] http://cwe.mitre.org/top25

978-0-7695-4742-8/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE.2012.23

user study of 47 novices. These novices applied software security test patterns to develop black box security tests for six functional software requirements from a publicly-available specification for electronic health records systems [8]. We compared the novices' plans to an oracle developed by a panel of researchers we assumbled, who developed a consensus around the security test plan.

The rest of this paper is organized as follows. Section II reviews the background for this paper. Next, Section III describes software security test patterns. Then, Section IV elucidates the methodology we used to conduct the user study. Section V presents the results of our study. Section VI presents the limitations of the study. Finally Section VII concludes the paper.

II. Related Work

This section describes the relevant related research to this paper.

A. Secure Software Development

Our software security test patterns are different than traditional black box testing techniques (e.g. those proposed by Beizer [9]). Software *security* testing entails that we validate not only that the system does what it should securely, but also that the system does *not* do what it is *not* intended to do [10]. To illustrate the difference, consider the following requirement: "The system shall provide the ability to send 250 character messages between users." The functional black box testing would result in a test plan that tests several variations on messages sent, such as trying a zero-length message, trying a message that is too long, sending the message to a non-existent user, or attempting to send with no database. Software security testing entails using this provided functionality in unintended ways, such as turning the message sending functionality into a spamming mechanism, sending malicious links to users within the system, or impersonating a different user. This unintended functionality is not often found in the requirements document unless the team has performed an explicit analysis of security requirements (i.e. [11–13]).

We do not intend the use of our software security test patterns to replace any existing methodology or technique. Austin and Williams found that there is no security methodology or technique which will find every type of vulnerability [14]. In light of this finding, we introduce our patterns as a new technique to help guide the software testers in one aspect of secure software development. Secure software methodologies, such as the Security Development Lifecycle (SDLC) [15] and OWASP's Comprehensive Lightweight Application Security Process (CLASP) [16], advocate considering security throughout the lifecycle. The concept of *building security in* prescribes that developers and testers consider system security from the outset of the project and design the system to be protected from malicious attack [17]. For example, towards the time of the product's release in the SDLC, an independent security team must finish a "security push." This team must close any unfixed security issues and review the system's threat models to ensure that all possible avenues of attack have been secured [16]. Both the SDLC and CLASP indicate security testing as one component or aspect of their methodologies [15], [16]. One important aspect of producing secure software is the execution of black box tests related to security, also known as penetration testing [18]. The success of current security assurance techniques that occur late in the product's lifecycle, such as penetration testing, vary based on the skill, knowledge, and experience of testers [18].

B. Security Patterns

Yoder and Barclaw were the first to introduce application security patterns [19], and since then, many more security patterns have been introduced (e.g. as summarized in [20]). Like the patterns introduced in Gamma, et al. [4], security patterns have the following components: Context, Problem, Forces, Solution, and Consequences. Software security patterns provide expert guidance about architecture and design for a secure system users. Our security *test* patterns deviate from these components because of the focus on testing rather than encapsulating design concepts.

Halkidis et al. [21] developed a methodology for characterizing the security risk for systems based on the number and type of security patterns present in those systems. Halkidis et al. evaluated two web-based applications, one which used some of the 13 security patterns from the Open Group Security Patern Technical Guide [22], and one group that did not use any patterns. Based on the results of an automated tool and prior experiences, Halkidis et al. determined the likelihood of certain attacks for each of these systems and then analyzed the ability of each security pattern to prevent exploits using these attacks. Halkidis et al. determined that the system that used the security patterns was significantly more resistant to attack than the system that did not.

Halkidis et al. also evaluated the same 13 patterns based on their ability to uphold ten security principles introduced by Gary McGraw [23]. The ten security principles include notions such as securing the weakest link, practicing defense in depth, and the principle of least privilege [24]. Halkidis et al. found that although there was no single pattern that could be used to uphold all ten principles, designers could achieve all ten principles with a proper combination of the patterns. These researchers carefully thought through how each pattern would or would not uphold each principle and summarized the results. Additionally, Halkidis et al. found that most patterns uphold between one an three principles and no pattern upholds more than six principles. We evaluated our security test patterns by the number of successful vulnerabilities we revealed using black box tests created with the patterns in earlier work [25]. In this paper, we also evaluate the accessibility of these test patterns for novice use in revealing vulnerabilities.

III. Software Security Test Patterns

Section A defines software security test patterns. Section B demonstrates how security test patterns are instantiated. Then, Section C provides an example pattern and an

overview of the initial patterns. Finally, Section D reviews our evaluation of the catalog.

A. What is a Software Security Test Pattern?

Design patterns were originally conceived by Alexander [26] in the field of building architecture, and tailored to software engineering by Gamma, et al. [4]. Alexander later introduced the notion of design pattern *languages* [5], which were tailored to software engineering by Coplien [7]. A pattern language is a collection of patterns that build on each other to generate a software system [5]. A pattern catalog is different than a pattern language in that a catalog is not necessarily complete or sufficient to develop or test an entire system.

For security test patterns, the "recurring security problem" is a vulnerability type or class of vulnerabilities. The "core solution" is a black box test that reveals those vulnerabilities. Finally, each test case that we produce with the pattern is one of the "millions of instantiations" possible with that pattern. A software security test pattern contains a template of a test case that exposes vulnerabilities, typically by emulating what an attacker would do to exploit those vulnerabilities. The parts of the pattern below that are in braces (e.g., <*insert object phrase*>) indicate instructions to the user on how to instantiate the pattern (see Section B).

We developed our patterns based on the CWE/SANS Top 25 Most Dangerous programming errors (see Section C). The "Targeted Vulnerability Types" component of the patterns is a list of the vulnerabilities types from the CWE/SANS Top 25 list that the pattern was based on.

The following is an example of a software security test pattern.

Pattern: Input Validation Vulnerability Tests

Keywords: Record, Enter, Update, Create, Capture, Store, Edit, Modify, Specify, Indicate, Maintain, Customize, Query, Receive, Search, Produce

Targeted Vulnerability Types: Cross-site Scripting, SQL Injection, Classic Buffer Overflow, Path Traversal, OS Command Injection, Buffer Access with Incorrect Length Value, PHP File Inclusion, Improper Validation of Array Index, Information Exposure Through an Error Message, Integer Overflow or Wraparound, Incorrect Calculation of Buffer Size, Race Condition, Uncontrolled Format String, NULL Pointer Dereference, Incorrect Conversion between Numeric Types, Untrusted Search Path, Use After Free, External Initialization of Trusted Variables or Data Stores, Missing Initialization

Test Procedure Template:

1. Authenticate as <*insert a registered user name*>.
2. Open the user interface for <*insert action phrase*>ing an <*insert object phrase*>.
3. Inject one random attack from the attack list[2] into a field of the <*insert object phrase*>.
4. Repeat the previous step for five attacks[3] from the attack list.
5. Repeat the previous two steps for five fields from the <*insert object phrase*>.

Expected Results Template:

- The system should gracefully inform the user that the input is invalid.
- The data store for the <*insert object phrase*> should remain intact.
- The system shall not reveal data that is not a part of this <*insert object phrase*>.
- No error messages should occur that reveal sensitive information about the system's configuration or architecture.

Example Natural Language Artifact: Requirement AM 02.04 - The system shall provide the ability to modify demographic information about the patient.

Example Test Procedure:

1. Authenticate as Dr. Robert Alexander.
2. Open the user interface for entering patient demographic information and create a new patient.
3. Inject one random attack from the attack list into a field of the demographic information.
4. Repeat the previous step for five attacks from the attack list.
5. Repeat the previous two steps for five fields from the patient demographic information.

Example Expected Results:

- The attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected and the user gracefully informed that their input is invalid.
- The data store for the demographic information should remain intact.
- No data should be revealed that is not a part of this patient's demographic information.
- No error messages should occur that reveal sensitive information about the system's configuration or architecture.

B. Instantiating Software Security Test Patterns

To instantiate a test pattern from our catalog, testers need a natural language artifact, such as a requirements statement. The content of the requirements statement can be used to guide the tester as to what types of vulnerabilities might be present, given the functionality provided by the requirement, and the type of black box security test(s) that should be executed to try to expose the vulnerabilities.

Traditional functional requirements are "shall" statements [27]. Requirements specifications like these typically conform to the following format: "*The system shall provide the ability to <action phrase> a <object phrase> <and/with/in supporting information>. <Supporting*

[2] Any attack list can be used, but for this paper we used a list of common attacks from http://neurofuzz.com.

[3] The choice of the number of tries for attacks is admittedly arbitrary. A security tester could execute as many attacks in as many fields as he or she desires. Some limit on the number of attacks will help in situations where testing a product is time-limited.

information>." The action phrase in these statements is typically an action that the system will perform on that data store, such as store, graph, view, print, or edit. The object phrase in these statements is most often a data store, such as a listing of users or a report regarding multiple data records for output. The supporting information in these statements provides additional information as to how, or when the system should achieve the key action phrase. Sometimes the supporting information is a separate sentence or can extend to an additional sentence.

Our methodology for creating application level security tests uses *key phrases* (key action phrase and key object phrase) and *supporting information* of the requirements statement to determine the type of security test that will most likely reveal vulnerabilities in the system. The first phrase that the tester comes to after reading "The system shall provide the ability to…" typically contains the key action phrase and is followed by the key object phrase. We call these phrases *key* because define the functionality the system has with respect to its environment. For example, consider a requirement that states, "*The system shall provide the ability to modify demographic information about the patient*," can be broken down as follows:

- **Key Action Phrase**: modify
- **Key Object Phrase**: demographic information about the patient
- **Supporting Information**: *none*

The phrase *modify* is the key action phrase. This key action phrase indicates that the funcionality involves modifying data. This new functionality may provide an attacker the ability to input malicious strings that can take the form of a cross-site scripting [28], SQL injection [29] or other input validation vulnerabilities. These attacks, if properly executed, have the potential to tamper with or reveal information from the *demographic information* object. Based on this intuition, a software tester should instantiate the Input Validation Test pattern (see Section III.A). The novice software tester would instantiate other test patterns from our catalog (described in Section III.C) depending on whether those patterns are signaled by the keywords in the requirements statement. In the example described above, the Audit and Force Exposure pattern, both described in Section III.C should also be instantiated.

A software tester who conducts our methodology repeats the procedure described in this section for every requirement found in the requirements specification, adding black box security test cases to a test plan as he or she goes. Depending on the order in which the tester encounters the requirements, a given requirement statement may produce none, one, or many test cases.

C. Overview of the Initial Pattern Catalog

We developed the set of initial test patterns using a grounded theory approach [6]. We examined the CWE/SANS Top 25, a list of the most dangerous application programming errors. We include the 23 vulnerabilities that CWE lists as being "on the cusp" as well, and we call this list the "CWE/SANS Top 25+." To create a pattern catalog using grounded theory, we first produce a black box test case that would successfully expose a vulnerability from the CWE/SANS Top 25+ using security expertise. We repeat this process until we cover all of the Top 25+ with at least one test case, and then we categorize and group the test cases by similarities in their test procedure and approach. Once the organization of the tests is complete, we extract the similar parts amongst all the groups and we obtain the patterns. Our software test pattern catalog (see Section C.2) should not be confused with a pattern language, in the way that Alexander and Coplien conceived of pattern languages. Instead, our *pattern catalog* is a collection of related patterns that can be instantiated within the same domain and contain the same elements (e.g. keywords, procedure template, an example of use, etc.) [4].

Using the technique described in this section, we obtained the following initial pattern catalog containing six patterns. Additional details, including a detailed description of each test type and the key phrases that the test type maps to can be found on our security test patterns wiki[4].

- **Input Validation Vulnerability Tests** – target vulnerabilities related to improperly validated user input, and are described more thoroughly in Section 2.
- **Force Exposure Tests** – expose functionality or information in a system that the user is unauthorized to use or see by recording the series of steps to get to the functionality and then repeating this series of steps without authorization.
- **Malicious File Tests** – upload a file that contains malicious scripting or otherwise would exploit a user who downloads that file by uploading some sample dangerous files and then trying to download them again.
- **Malicious Use of Security Functions Tests** – exploit or misuse security functionality such as passwords, encryption, hashing, changing passwords, etc.
- **Dangerous URL Tests** – inject a URL for the purposes of phishing or spamming a user.
- **Audit Tests** – check that all actions performed on data that is sensitive or protected are recorded in a human-readable format.

D. Initial Evaluation

In prior work [25], we created test cases based upon these patterns using 284 functional requirements from a public specification [8] to generate 137 black box tests. We then executed these tests on each of five electronic health record

[4] http://securitytestpatterns.org

systems: OpenEMR[5], ProprietaryMed[6], WorldVistA[7], Tolven[8], and PatientOS[9]. These systems are currently used to manage the clinical records for approximately 59 million patients, collectively: Out of the 685 total test executions, 253 (37%) revealed vulnerabilities in the five systems. Also, our evaluation shows that using our patterns reveals vulnerabilities typically missed by automated penetration testing and static analysis (e.g. design flaws). An undergraduate student with minimal security experience also executed the test plan on our study subjects and achieved the same results, indicating that novices can effectively *use* the test plan. In this paper, we evaluate the ability of novice software testers to *create* a test plan using our six initial test patterns.

E. Tool Support: STPI

We have implemented Security Test Pattern Instantiator, or STPI, a requirements parsing tool that builds upon the Stanford Parser libraries [30]. STPI helps software testers quickly and properly instantiate our patterns over a set of requirements to develop a black box security test plan. A running copy of STPI is available from our security test patterns website[10]. STPI uses the natural language processing engine within the Stanford Parser to extract the key phrases described in Section III.B. STPI presents the user with one or more phrases parsed from the entered natural language specification, the *default* key phrase, and an *other* box for manually entering their own key phrase. After selecting the requisite key phrases from the parsed natural language requirements statement, STPI presents our security test pattern catalog. If a keyword that is contained within the pattern was found in the correct key phrase of the natural language statement, then that pattern is selected by default. After selecting requisite key phrases and the applicable patterns, the user clicks "Select and Edit Test Cases," and STPI provides the user with generated test cases. The test cases contain the *Test Procedure Template* and *Expected Results Template* from the selected patterns with the key phrases the user selected automatically filled in. After instantiating a pattern, the user can save the results to an editable black box security test plan. STPI also allows the user to customize or create patterns, handle sets of requirements, automatically parse requirements, and export their black box test plan. We present a screenshot of using STPI to parse a requirement and generate a black box test in Figure I.

IV. STUDY DESIGN

We conducted a user study of 21 graduate and 26 undergraduate students who have relatively low security training or expertise (novices). When the study was conducted, the undergraduate students had no training in how to conduct security analysis. The graduate students had begun a course in software security, but had not yet learned how to perform security testing. These novices applied software security test patterns to develop black box security tests for six functional software requirements from a publicly-available specification for electronic health records systems [8]. Using the Goal-Question-Metric template, as proposed by Basili et al. [31], we expand our goal statement:

> Analyze the **process of using software security test patterns to develop security test cases**
> for the purpose of **evaluation**
> with respect to **effectiveness for providing an accessible security test generation technique**
> from the perspective of **the novice software tester**.

Our goal statement results in several research questions, each of which have one or more associated metrics or measurements that can be used to answer the questions as they pertain to the goal [31]. For our patterns to be effective, the patterns and their application must be *accessible*, that is, they must be easy to use and easy to understand. We elicit three research questions regarding the accessibility of the patterns and technique:

RQ1. Do the novices use the patterns to generate a similar black box test plan as the experts could?

RQ2. How time consuming is the process of instantiating patterns into black box tests?

RQ3. Do novices find the patterns and their use accessible after they have completed the study?

We present the metrics and measurements associated with each of these research questions when we answer them in Section V. The rest of this section is organized as follows. Section A describes the operation of our research tool. Section B describes how we collected the expert consensus. Then, Section C describes our user sample and response rate. Finally, Section D describes the requirements set.

A. Research Instrument: STPIPrime

We customized STPI (see Section III.E) for this study to create STPIPrime, which helped gather the data from the novices who participated in our study. For the case study, we removed the ability to customize patterns or handle requirements set, and hard-coded STPIPrime so that novices could only access the requirements used in this study. We also modified STPIPrime to store the selections the novice made for each key phrase, as well as the patterns the novice chose to instantiate for each requirement in a database table. We also augmented STPIPrime with an audit log for each novice as they proceeded through the exercise which marked a timestamp for each action the novice took. The novices were not aware that we were tracking their times. We also built a survey tool into STPIPrime. The survey asked the novices' opinion of the the patterns, their application, and the amount of years of experience the novice had in software engineering and software security. We present the questions of this survey with the results in Section V.C.

[5] http://oemr.org/
[6] ProprietaryMed was developed by an organization that wishes to keep the identity of their product confidential.
[7] http://worldvista.org/
[8] http://tolven.org/
[9] http://patientos.org
[10] http://securitytestpatterns.org

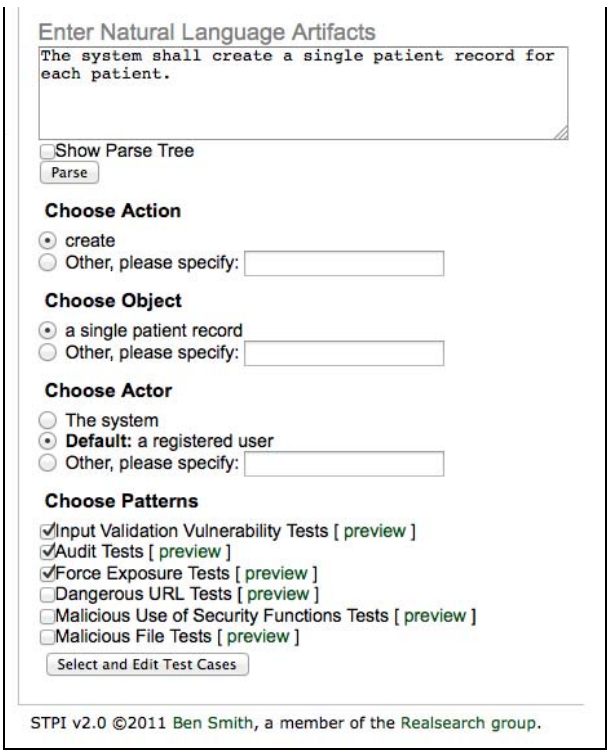

Figure I. Using STPI to Parse a Requirement

B. *Establishing Expert Consensus*

The key phrases selected by a software tester in each requirement, as well as which security test patterns to instantiate is a subjective decision. As such, before we conducted our study of the novices, we established a baseline consensus on which pattern(s) should be instantiated for each requirement. We formed an expert panel of six doctoral students and one undergraduate student. Many of the students in this panel have conducted extensive research in software security. We presented the experts with each requirement, and asked them to select the key object phrase, actor phrase and action phrase. We also asked the experts to select which test patterns of the six they would instantiate. Using the Delphi Method [32], after making each decision, and writing it down in secret, the experts read their decision aloud and discussed the reasons for their choices. Then we asked everyone to reconsider their choices and vote again. We repeated this process until everyone had the same vote and established a consensus. The experts made their choices manually, without using `STPIPrime`. The process of forming consensus among the experts took approximately three and a half hours. We recorded the experts in their discussions about each decision, and transcribed the entire session. Additionally, we kept a record of every expert's vote on each decision, including the final vote that resulted in a consensus. The Delphi Method allows for multiple definitions for a stopping point when we could claim that the experts had consensus. We chose to ask our experts to form a unanimous verdict because many of the discussions regarding the correct choice of key phrase or the applicable pattern would only come out when a group of the experts were trying to convince one dissenter.

C. *User Subject Groups*

We called for participation from novices in an undergraduate software engineering course and a graduate software security course, both at North Carolina State University in the United States. We present an overview of the sample sizes and response rates for each group in Table I. The second column in Table I presents the number of complete responses we have from each group. The third column in Table I presents the number of novices who were offered the chance to take the study. We offered novices from the undergraduate course 1 extra percentage point on their course grade if they completed the study. Fifty-nine (59) of 68 students were in attendance the day we presented the study.

Table I. Response Rate and Sample Sizes

	Responses (Sample Size)	**Population Size**	**Response Rate**
Graduate	21	26	81%
Undergraduate	26	68	38%
Overall	47	94	50%

D. *Requirements Set*

In 2006, the Certification Commission of Healthcare IT (CCHIT) defined 284 certification criteria focused on the functional capabilities that should be included in EHR systems [33]. In this paper, we chose to instantiate our test cases using the CCHIT certification criteria [8] because our previous work (summarized in Section III.D) was based on the CCHIT criteria. The novices and experts were each given nine decisions (three key phrases, six patterns) to make about each requirement in the first six CCHIT requirements. We present the first six CCHIT requirements in Table II.

Table II. User Study Requirements Set

Requirement ID	Requirement Text
AM 01.01	The system shall create a single record for each patient.
AM 01.02	The system shall associate (store and link) key identifier information (e.g., system ID, medical record number) with each patient record.
AM 01.03	The system shall provide the ability to store more than one patient identifier for each patient record.
AM 01.04	The system shall provide a field which will identify patients as being exempt from reporting functions.
AM 01.05	The system shall provide the ability to merge patient information from two patient records into a single patient record.
AM 02.01	The system shall provide the ability to include demographic information in reports.

V. RESULTS

In this section, we provide results from our study that help answer the research questions we outlined in Section IV.

A. Level of Agreement

RQ1. Do the novices use the patterns to generate a similar black box test plan as the experts could?

Metrics: Kappa scores for agreement between novices and experts for key phrases and applicable patterns.

The **Fleiss Kappa Score** is a measurement of inter-rater agreement amongst a fixed number of raters [34]. We use the Kappa score to measure the amount to which a group of people agree on the key phrases and applicable patterns for a requirement. We also use the Kappa score to measure the amount to which the novices agree with the experts. Each of the raters gives an answer on a set of *n* subjects, and can decide among many categorical options (e.g. a, b, or c). The Kappa score represents the extent to which the observed amount of agreement exceeds what would be expected if all raters made their ratings completely randomly. A Kappa score of 1.0 indicates that all *k* raters agreed on each of the subjects in the sample. A Kappa score of zero or less indicates that all *k* raters agreed less than would be expected by random chance. A Kappa score is also assigned a p-value, which represents the probability that this sample's Kappa is a chance occurrence. A Kappa with a low p-value means that the population this sample was taken from is likely to have a similar Kappa score. For the purpose of this paper, we consider a Kappa of 0.2 or higher to be sufficient to say that an agreement exists, as indicated by Landis and Koch [35]. We consider a p-value below the 0.05 level as statistically significant.

First, we compare the novices individually to the experts. We calculated the Kappa score of each novice's agreement with the expert conensus on all 54 decisions they made for this study. Figure II displays the distribution of Kappa scores for the 47 novices' agreement with the expert consensus. With an average Kappa of 0.303 and an interquartile range of 0.269 to 0.354, we can say that the novices had a fair agreement with the experts overall. We also calculated the p-values for each of these Kappa scores and found that only two Kappa scores had a p-value above our acceptable 0.05 level.

Next, we compare the novices agreement with **each other**. Using the Fleiss Kappa score for 54 subjects and 47 raters, we arrive at a Kappa of 0.547 with a p-value of less than 0.0001. By looking at only the decisions about key phrases, we can see that the novices strongly agreed with each other, with a Kappa of 0.673 and a p-value of less than 0.0001. The novices shared fair agreement with whether each pattern was applicable with a Kappa of 0.246 and a p-value of less than 0.0001.

In summary, we found fair agreement between each novice and the experts for the 54 decisions that each group made for this study. The novices also express fair agreement among themselves for the 54 decisions they made in this study.

Indications: Novices using security test patterns seem to

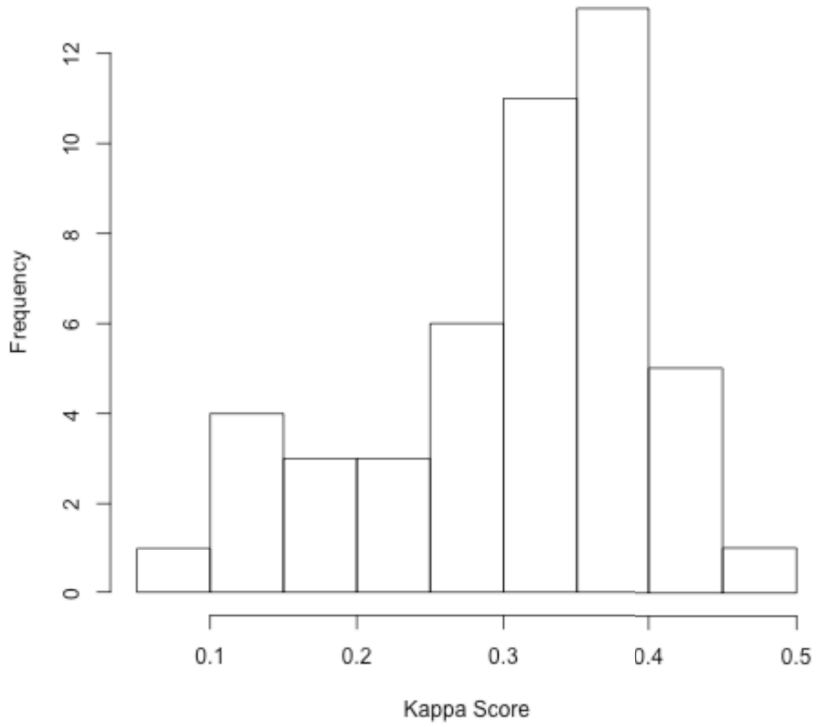

Figure II. The Distribution of Kappa Scores for Novices Agreement with the Experts

make somewhat similar decisions about which patterns are applicable as other novices do. Novices also make similar decisions about which patterns are applicable as the experts.

B. Novice Times

RQ2. How time consuming is the process of instantiating patterns into black box tests?

Metrics: Average times of completion on each requirement for each novice, average total time to complete exercise, and average time per requirement.

As we described in Section IV.C, `STPIPrime` marked timestamps down to the second for each action the novices completed. In particular, we marked the time when a novice is first shown a requirement, and the time when the novice has saved the generated test cases for a requirement. To analyze these data, we manually inspected each novice's audit log and annotated the timestamps of the sequence of events to group the novice's actions into several groups: 1) the time for the completion of parsing each individual requirement; 2) "ramp up": the time before the novice began parsing requirements, which includes reading the instructions, interacting with the demo parser, any false-starts the novice made with the requirements set; and 3) "cool down": the time after the novice had completed parsing all requirements, but had yet to log out of the system, which includes time spent taking the survey.

After we had finished manually analyzing these data, we had a time for completion in seconds for each novice on each requirement, as well as the ramp up and cool down phases. We took the average of these times for n=47 novices and present the result in Table 8. We calculated the row labeled "Reqs. Only" by subtracting the timestamp when the novice started the first requirement from the timestamp when the novice finished the last requirement. We calculated the "Time/Req." row by dividing the "Reqs. Only" row by n=6 requirements, as presented in Table III.

- Novices spent on average 29 minutes 49 seconds completing the entire user study.
- Novices spent on average 14 minutes 9 seconds to generate a security test plan for six functional requirements.

We also assigned an ordered value for each of the first six phases of the exercise that we observed. In this study, "1" was the "ramp up" time, "2" was requirement AM 01.01, and so on. If we fit a line to this relationship, we can analyze the effect of the change in phase has on the novices' average time to complete that phase. Using this technique, we were able to see that in our sample, novices spent less time analyzing each requirement as they proceeded through the phases (Simple Linear Regression, $R^2 = 0.1792$). This decrease in time per requirement seems to indicate that novices can make decisions more quickly as they gain experience.

We analyzed the number of test cases produced by each novice as a dependent variable and found that the number of tests a novice produced increased linearly with the amount of time that novice spent in the "Ramp Up" phase of the exercise (Simple Linear Regression, $p<0.01$, $R^2=0.1385$). We did not find that the number of tests produced by each novice was correlated with the total time for the study or the total time the user spent on the requirements. However, we found that novices produced 17 tests on average for these six requirements, or approximately 2.8 test cases per requirement.

Indications: Novices spent on average 29 minutes 49 seconds to complete the exercise, and spent 14 minutes 9 seconds (or 2 minutes 21 seconds/requirement) on the six requirements. Novices took a decreasing amount of time to analyze each requirement as they proceeded through the study. Novices produced at least 15 tests in the exercise, or approximately 2.5 test cases per requirement.

C. User Survey

RQ3. Do novices find the patterns and their use accessible after they have completed the study?

Table IV presents the questions and the mean/median answer by the novices to the survey that was administered within `STPIPrime` , as described in Section IV.A. On the first two questions, the answer was free-form text input (with input validation), so we report the mean for a continuous variable. For the remaining questions, the answer was a drop-down box selection between 1 and 5. For these answers, the appropriate summary statistic is the median due to the scale type [37].

We note that despite the fact that novices thought the exercise was time consuming (with an overall median of 3), they also thought it was useful for revealing tests

Table III. Average Times in Min:Sec for Requirement Completion

Phase	Overall	Undergrad	Graduate
Ramp Up	12:18	10:35	14:26
AM 01.01	4:55	4:38	5:17
AM 01.02	2:32	2:58	1:59
AM 01.03	1:51	1:06	1:34
AM 01.04	2:30	3:05	1:47
AM 01.05	1:06	1:08	1:04
AM 02.01	1:12	1:31	0:49
Cool Down	3:20	4:36	1:47
Total Time	29:49	30:40	28:44
Reqs. Only	14:09	14:09	14:09
Time/Req.	2:22	2:22	2:22

(median=4). Novices also felt that the patterns would be useful for revealing vulnerabilities (median=4). We note also from the results of the survey that the novices reported lower amounts of experience in software development and software security than the experts did on average. We could not obtain statistical significance on the results from the questions in the user survey, meaning that the sample may not be representative of the population of novices.

Indications: The novices thought that the technique was time consuming, but that the technique and the patterns were useful for discovering vulnerabilities.

VI. LIMITATIONS

One limitation of applying our software security patterns is that a software tester must have an available requirements specification that clearly indicates how the system should be used. Additionally, this sample of graduates and undergraduates may not be representative of software testers at a development organization. Other studies should test other samples of software testers, especially those in industry. We only used six requirements from one specification. Other requirements from this requirements specification and other requirements specifications may produce differing results in terms of any of the measurements we obtained during this study. Although we demonstrated that these patterns are effective at revealing security vulnerabilities in earlier work, the novices in this study did not execute their test plans on any resultant system and therefore we do not know how effective their test plans would be for revealing vulnerabilities. The coding of the recorded data and the analysis of the data for statistical relationships are both subject to human error. The statistical results from this study may only be valid for this study. Our interpretation of the meaning of the results is subjective. Although Landis and Koch indicate [35] that a Kappa of 0.2 or higher indicates "fair agreement", Gwet has explained [38] that these assigned meanings may be more harmful than helpful in understanding Kappa scores because the number of categories and subjects will affect the magnitude of the value from experiment to experiment.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we conducted a user study of 21 graduate and 26 undergraduate students who have relatively low security training or expertise (novices). These novices used a research tool we developed to develop a black box security test plan using six initial security test patterns based on six requirements from a publicly available specification. Our goal was to evaluate the ability of novices to effectively generate black box tests by accessing security expertise contained within security test patterns. We found,

- Novices using security test patterns will each make similar decisions to the experts when developing a black box test plan using security test patterns.
- Novices will also make similar decisions with each other when developing a black box test plan using security test patterns.
- Novices spent 29 minutes 49 seconds to complete the exercse on average, with 14 minutes 9 seconds of that parsing the requirements. Novices produced 17 tests on average while completing the exercise, for an average of 2 minutes 21 seconds per requirement and 2.5 tests per requirement.
- Novices reported that they felt the exercise was time consuming, but that it would be useful for finding vulnerabilities and that the patterns they used in this study would also be useful for discovering vulnerabilities.

In summary, we provide empirical evidence that novices can effectively access the knowledge contained within our software security test patterns to generate black box security test plans.

In future work, we will extend the range of this sample to software testers in industry. We will also have software testers using the research tool we developed on a larger set of requirements and more varied requirements. Our theory is that software testers who are using the technique to develop a security test plan that they will use on an industrial system will produce differing results than software engineering students who are completing the exercise as a part of their

Table IV. Summary Statistics for Survey Responses in the Sample

Question	Novices Overall	Education Level		Experts
		Grad.	Undergrad.	
Sample Contains (n=?)	**47**	**21**	**26**	**7**
How many months of software development experience have you had*?	18.6	27.1	11.7	63.1
How many months of software security experience have you had before taking this course?	1.79	2.29	1.39	12.3
How much software engineering education have you had (1 year or less, 2 years or less, …, more than 5 years)?	3	2	3	5
Please rate your overall experience with this exercise (1 = Terrible, 5 = Excellent).	3	4	3	4
How useful do you think this technique would be for discovering vulnerabilities (1 = Terrible, 5 = Excellent)?	4	4	4	4
How time-consuming do you think this exercise would be for non-expert software testers (1 = Not too time consuming, 5 = VERY time consuming)?	3	3	3	4
How useful do you think the patterns you used today are for revealing vulnerabilities (1 = Not at all useful, 5 = Very useful)?	4	4	4	4
* This does include development experience in academic courses.				

educational experience. In future work, we will also expand the pattern catalog to include a broader array of a software security test procedures.

ACKNOWLEDGEMENT

This work was supported by an IBM PhD Fellowship. We would also like to thank John Slankas for his help in testing `STPIPrime` before its release to the subject group. We would also like to thank the NC State Realsearch group for their helpful comments in revising this paper.

REFERENCES

[1] M. L. Kelly, "Cyberwarrior Shortage Threatens US Security," NPR Morning Edition, http://www.npr.mobi/templates/transcript/transcript.php?storyId=128574055, 2010.

[2] R. Lemos, "Security lessons still lacking for computer science grads," *InfoWorld Magazine*, http://www.infoworld.com/t/application-security/security-lessons-still-lacking-computer-science-grads-769, 2010.

[3] K. Evans and F. Reeder, "A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters," Center for Strategic and International Studies, http://csis.org/files/publication/101111_Evans_HumanCapital_Web.pdf, 2010.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison Wesley Longman Publishing Company, 1995.

[5] C. Alexander, *A Pattern Language: Town, Buildings, Construction*. Oxford, UK: Oxford University Press, 1977.

[6] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory; Strategies for Grounded Research*. New York, NY: Aldine de Gruyter, 1967.

[7] J. Coplien, *Software Patterns*. New York, NY, USA: SIGS Books & Multimedia, 2000.

[8] "CCHIT Certified 2011 Ambulatory EHR Certification Criteria," The Certification Commission for Health Information Technology, http://www.cchit.org/sites/all/files/CCHIT%20Certified%202011%20Ambulatory%20EHR%20Criteria%2020100326.pdf, 2010.

[9] B. Beizer, *Softwar Testing Techniques. 2nd Edition.* London: International Thomson Compute Press, 1990.

[10] R. C. Martin and G. Melnik, "Test and Requirements, Requirements and Tests: A Möbius Strip," *IEEE Software*, vol. 25, pp. 54-59, 2008.

[11] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeg, "Security Requirements Elicitation: A Framework for Representation and Analysis," *IEEE Transactions of Software Engineering*, vol. 34, pp. 133-153, 2008.

[12] N. R. Mead and T. Stehney, "Security quality requirements engineering (SQUARE) methodology," in *Software Engineering for Secure Systems (SESS)*, St. Louise, Missouri, USA, 2005, pp. 1-7.

[13] A. van Lamsweerde, "Elaborating Security Requirements by Construction of Intentional Anti-Models," in *International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, 2004, pp. 148-157.

[14] A. Austin and L. Williams, "One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques," in *Emperical Software Engineering and Measurement (ESEM)*, Banff, Alberta, Canada, 2011.

[15] J. Gregoire, K. Buyens, B. D. Win, R. Scandariato, and W. Joosen, "On the Secure Software Development Process: CLASP and SDL Compared," in *International Conference on Software Engineering. Software Engineering for Secure Systems (SESS 2007)*, Minneapolis, MN, USA, 2007, pp. 1-7.

[16] S. Lipner, "The Trustworthy Computing Security Development Lifecycle," in *20th Computer Security Applications Conference*, Tuscon, Arizona, 2004, pp. 2-13.

[17] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[18] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, pp. 84-87, 2005.

[19] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," *Urbana*, vol. 51, p. 61801, 1998.

[20] T. Heyman, K. Yskout, R. Scandariato, and W. Joosen, "An analysis of the security patterns landscape," in *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, 2007, p. 3.

[21] S. T. Halkidis, N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Architectural Risk Analysis of Software Systems Based on Security Patterns," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 3, pp. 129-142, Sep. 2008.

[22] B. Blakley and C. Heath, "Members of the Open Group Security Forum," *Security Design Patterns, Open Group Technical Guide*, 2004.

[23] S. T. Halkidis, A. Chatzigeorgiou, and G. Stephanides, "A Qualitative Evaluation of Security Patterns," in *Information and Communications Security*, vol. 3269, J. Lopez, S. Qing, and E. Okamoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 132-144.

[24] J. Viega and G. McGraw, *Building secure software: how to avoid security problems the right way*. Addison-Wesley Professional, 2001.

[25] B. Smith and L. Williams, "Systematizing Security Test Planning Using Functional Requirements Phrases," North Carolina State University, TR-2011-5, ftp://ftp.ncsu.edu/pub/unity/lockers/ftp/csc_anon/tech/2011/TR-2011-5.pdf, 2011.

[26] C. Alexander, *The Timeless Way of Building*. Oxford, UK: Oxford University Press, 1979.

[27] K. E. Wiegers, *Software Requirements, 2nd Edition*. Redmond, WA: Microsoft Press, 2003.

[28] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 171-180.

[29] W. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for NEutralizing SQL injection attacks," in *International Conference on Automated Software Engineering*, Long Beach, CA, 2005, pp. 174-183.

[30] M. P. Marcus, M. A. Marcinkeiwicz, and B. Santorini, "Building a large annotated corpus of Enligh: The Penn Treebank," *Journal of Computational Linguistics*, vol. 19, pp. 313-330, 1993.

[31] V. R. Basili and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments," *Software Engineering, IEEE Transactions on*, vol. 14, no. 6, pp. 758–773, 1988.

[32] H. A. Linstone and M. Turoff, "The Delphi Method: Techniques and Applications," *Technometrics*, vol. 18, no. 3, p. 363, 2002.

[33] HHS Press Office, "ONC Issues Final Rule to Establish the Temporary Certification Program for Electronic Health Record Technology," 2010.

[34] J. L. Fleiss, "Measuring nominal scale agreement among many raters.," *Psychological Bulletin*, vol. 76, no. 5, p. 378, 1971.

[35] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, pp. 159–174, 1977.

[36] J. F. Box, "Guinness, Gosset, Fisher, and Small Samples," *Statistical Science*, vol. 2, no. 1, pp. 45-52, Feb. 1987.

[37] S. S. Stevens, "On the Theory of Scales of Measurement," *Science*, vol. 103, no. 2684, pp. 677 -680, Jun. 1946.

[38] K. Gwet, "Handbook of inter-rater reliability , The definitive guide to measuring the extent of agreement among multiple raters," *Advanced Analytics, LLC, Gaithersburg, MD*, 2010.