World Wide Web 1 (1998) 193–208

Web-based testing: A study in insecurity

Joel C. Adams [a] and Aaron A. Armstrong [b]

[a] *Department of Computer Science, Calvin College, Grand Rapids, MI 49546, USA*
E-mail: adams@calvin.edu
[b] *Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA*
E-mail: aarmst@umich.edu

Much has been written about the distinct ideas of *on-line testing*, in which students take tests and receive immediate feedback on their performance; *distance learning*, in which telecommunication technology extends the traditional classroom beyond the classroom walls; and *multimedia*, in which text, sound and graphics are integrated within an application. By using the World Wide Web (WWW) as a medium for administering on-line tests, immediate feedback can be provided, tests can be administered at a distance, and multimedia resources can be conveniently incorporated into questions. This paper describes **Eval**, an undergraduate research software prototype in which we explore security issues related to using WWW as a testing medium.

1. Introduction

At most high schools and universities, the act of taking a test is little different today than it was 100 years ago. Students bring pens or pencils to a classroom, receive a set of test questions written on paper, and begin to answer them, again on paper. When done with the test, students submit their test papers for evaluation, and after some delay (ranging from days to weeks), are informed of how they scored. There are several drawbacks to this traditional approach to testing:

- One drawback is that an important educational opportunity is squandered because of the delay between the time students answer a question and the time they receive feedback regarding the correctness of their answers. It has long been known from associative learning studies that the sooner feedback is received for a response, the more rapidly learning occurs, even in the simplest models of learning [Harker 1956; Perin 1943]. If after answering a question, a student could *immediately* be told the correct answer to that question, then the association between that question and its correct answer would be strengthened. This in turn would make testing truly educational, instead of merely evaluative, since students could leave a test with a deeper level of understanding of its material than they had prior to taking it.
- A different drawback emerges in the context of *distance learning* programs [Derringer 1985; Marker and Ehman 1990; Raval 1993; Winn *et al.* 1990] in which telecommunications technology is used to permit students at one university (or even at home) to take a course offered at another university. In such programs, the traditional classroom is being replaced by the *distributed classroom*, the traditional university replaced by the *distributed university*. Using traditional testing methods in these new environments is clumsy, at best.
- The final drawback we will mention is the amount of time that conscientious educators devote to correcting the thinking of their students. Educators who write corrections on a test paper and/or meet with students one-on-one devote significant amounts of time outside of the classroom providing students with feedback regarding *why* what they did was incorrect or correct. Educators who opt for the alternative approach of devoting a class period to "go over the test" may lose several class periods of instruction each semester.

Online testing holds the potential to eliminate these drawbacks. More precisely, an on-line testing system can be designed to:

(1) provide a student with immediate feedback tailored to their particular response to a question,

(2) provide built-in support for distance education, and

(3) automatically score and report the mistakes on student tests, allowing educators to focus on teaching, leaving mechanical tasks to the machine.

This paper describes **Eval** (for **E**ducational **v**erification **a**nd **l**earning), a prototype on-line testing system that seeks to provide these capabilities, using the World Wide Web [Berners-Lee *et al.* 1992] as a delivery medium. The paper details how our design evolved over time in response to security issues raised by using the Web as a testing medium. The paper has the following structure: section 2 presents a detailed description of our goals for the system, discusses how these goals distinguish **Eval** from other on-line testing systems, and illustrates how our prototype realizes these goals. Section 3 presents our initial design and implementation of the system, and details its evolution in response to security concerns unique to the Web. Section 4 provides a discussion of different means of providing security in the Web environment, and section 5 finishes with our plans for the future. Documentation about our prototype can be found at `http://cs.calvin.edu/eval/`.

2. System objectives

At the outset of this project, our primary objective was to build a prototype on-line testing system providing the following capabilities:

1. Distance-learning support, so that a student can take a test from anywhere on the Internet.
2. Cross-platform compatibility with the windowing systems of each of the three most commonly used operating systems (MacOS, MS-Windows, and UNIX/X).
3. A convenient, easy-to-use *test taking tool* with a mouse-driven graphical user interface (GUI) that allows students to take multiple-choice tests on-line and receive immediate feedback each time they answer a question.
4. An intuitive *test editing tool* with a GUI that allows a professor to construct such tests as easily as possible. In addition, this tool allows a professor to
   (a) associate an *explanation* with each multiple-choice answer, to provide answer-specific feedback, and
   (b) incorporate *multimedia resources* (graphics, audio, movies) into test questions.
5. Automated grading of tests, plus reporting of scores and statistics to the instructor.
6. An effective security system against intruders and external tampering.

In the remainder of this section, we discuss how these goals distinguish **Eval** from other efforts to computerize testing.

2.1. Related efforts before 1994

Computing has changed significantly since we began this project in early 1994. At that time, the World Wide Web was just beginning to explode in popularity, and the platform-independent language *Java* was not yet widely known or publicized. The available systems for on-line testing were by and large *dedicated systems* designed to run in a homogeneous computing environment. One difference between **Eval** and on-line testing systems such as LXR*Test [LXR*Test 1994] or those discussed in [Leclercq *et al.* 1993; Vockell and Hall 1989] is those system's lack of support for our cross-platform compatibility objective.

Distance learning environments are almost by definition heterogeneous, and so our objective of providing a multi-platform system was a high-priority from the outset. Early in 1994, our first thought was to develop **Eval** using a multimedia authoring package such as *Authorware* from Macromedia [Macromedia 1996] or *ToolBook* from Asymetrix [Asymetrix 1996]. However, we were in 1994 unable to find any multimedia authoring system that provided both the built-in networking capabilities needed to achieve our distance learning objective and the support for the MacOS, MS-Windows and UNIX/X windowing systems needed to achieve our cross-platform compatibility objective.

2.2. Related efforts since 1994

Since 1994, the World Wide Web (WWW) has exploded in size and popularity. Two reasons for its unprecedented growth are:

(1) its provision of the uniform resource locator (URL), a simple scheme for addressing and accessing text, graphics, sound or video elements anywhere on the Internet,

(2) its support for HTML *forms* and *CGI-scripts*, which provide a simple, built-in mechanism for performing networked transactions across the Internet,

(3) the availability of free or inexpensive WWW browsers for most platforms, including MacOS, MS-Windows, and UNIX/X.

These features led us to explore the use of the World Wide Web as a *medium* for performing on-line testing. More precisely, forms and CGI-scripts provide the necessary support for building a test on-line, URLs and the hypertext transport protocol (HTTP) on which WWW is based provide a straightforward means of achieving our distance learning and multimedia objectives, and the availability of browser software for each of our target platforms means that such a system can be used in a platform-independent manner.

We have not been alone in recognizing the potential of the Web for on-line instruction and evaluation. Related efforts in this area include:

- **QM Web** is a commercial product from Question Mark Computing [Question Mark 1995] that provides a means of using the World Wide Web to administer tests consisting of multiple choice, multiple response, text, and selection questions. QM Web only permits questions to be created in the Windows environment, and thus does not meet our goals for multiple-platform compatibility. Question Mark Computing is up-front about the security problems that are present in QM Web, which we believe are common to many web-based testing systems.
- **ToolBook II** is a commercial product from Asymetrix Corporation [Asymetrix 1996] that provides a set of tools by which tutorials and tests can be created. The tools provide support for multiple-choice, true-false, completion, matching, and ordering questions. By writing one's test as an "Internet WebBook", ToolBook will implement the test through a mixture of HTML forms and Java applets, which will then run on any Java-compliant WWW-browser. While this new version of ToolBook meets many of our objectives, it does require that the *creator* of a test work in the Windows environment, since that is ToolBook's native platform. It thus does not quite meet our requirements for multi-platform compatibility.
- **QUIZIT** [Tinoco *et al.* 1997] is an academic tool developed at Virginia Tech. Like other recent systems, it uses

WWW to distribute content, including on-line quizzes, and thus permits students to take a quiz on any platform for which a web-browser is available. (The database interaction "modules" that make up QUIZIT must reside on a UNIX server.) Instructors must use a text or SGML editor to author tests using QUIZIT Markup Language, an application of the Standard General Markup Language (SGML).

QUIZIT's designers have addressed several of the potential security risks of a Web-based testing system [Johnson 1997; Tinoco *et al.* 1996]. However, QUIZIT requires that test authors be familiar with how a tagged language works [Tinoco *et al.* 1997], and requires that they use a text or SGML editor to author quizzes.

- **WebCT** [WebCT 1997] is a commercial product developed at the University of British Columbia that provides a sophisticated set of WWW course tools by which instructors can convene their courses on-line, including the administration of tests. All interaction with the system occurs via a web-browser, making the system platform-independent to the end user. (At present, the WebCT server must reside on a UNIX system.) One difference between the testing facility of WebCT and **Eval** is that a student taking a WebCT test must check back at a later time to determine their score, whereas our objective is to provide a student with *immediate feedback on a question-by-question basis*, so that students knows their scores as soon as they complete their tests.
- The Medical Council of Canada has recently begun building a testing system for delivery of the exam to license physicians in Canada [Miller 1998]. This system delivers an *adaptive test* [Wainer 1990], in which each physician candidate begins with a question at the same level of difficulty. If the candidate answers this question correctly, they next see a more difficult question; but if they answer it incorrectly, they next see an easier question. Adaptive tests are useful in testing situations where one wishes to establish an individual's general level of competence (i.e., identify incompetent physicians), because such tests can do so with a minimal number of questions.

  Such a system demands high security measures. Tests will be administered in a proctored computer laboratory. Candidates will have to show proof of identify to enter the room, and receive an ID card containing a user name and password. To use the system, a candidate's user name and password *plus* the user name and password of the proctor must be entered. The system will use a combination of Perl and Javascript to present a customized full-screen browser. Each question is displayed in a separate window. When a question is displayed, the window of any preceding question is closed (preventing a candidate from returning to an already answered question). A wide variety of additional security measures are planned, including the disabling of all menus, the disabling of browser caching, and authentication through *cookies* (see section 4.2.1).

  This system has a comprehensive, well constructed security plan. Its evolution has in many ways paralleled the development of our system Eval. However, the purpose of this system is to administer a particular standardized national physician-certification exam. By contrast, our goal is to create a general-purpose testing system that any high school or university can use to create and deliver a variety of on-line tests.

This list is far from exhaustive, but it does provide a representative sample of the tools for on-line testing via WWW that are presently available or under development.

3. Design and implementation

As noted above, we discovered early in the project that the World Wide Web provided a means of achieving our multimedia, distance-learning, and cross-platform compatibility objectives.

The remaining open question for our prototype to answer was this: *Can a WWW-based on-line testing system be constructed that provides adequate security?* Security holes can result from flaws in a specific system [Garfinkel and Spafford 1996] or they can result from failure to adhere to generic security standards [Icove *et al.* 1995; Russell and Gangemi 1991]. In this section, we present the evolution of our design and implementation in seeking to answer this question.

3.1. Initial design

Being new to web-based systems in 1994, our initial design consisted of three basic components:

(1) a *test database*, a database in which to store the data of one or more tests,

(2) a *test-taker*, a collection of CGI-scripts providing a web-based "front-end" to the test database, allowing a student to take a test, and

(3) a *test-editor*, a collection of CGI-scripts providing a second web-based "front-end" to the test database, allowing an instructor to create or modify a test.

Figure 1 illustrates this initial design of our system.

3.1.1. The test database

To implement our system, we chose a departmental Sun SPARC-2 that was lightly used at the time. For our test database, we choose the freeware GNU database manager (gdbm) from the Free Software Foundation [GNU 1997] which is available for a variety of different hardware platforms.

It is worth noting that like many freeware database packages, gdbm provides virtually *no security features*. Since high schools and universities often use freeware, it was our

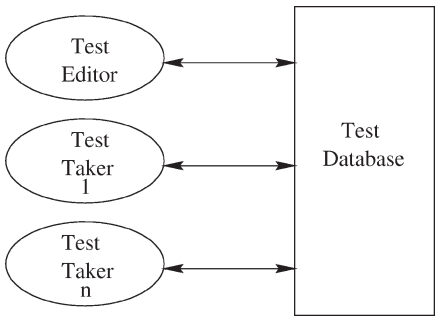

Figure 1. Initial design.

intent to build a secure on-line testing system that did not rely on a secure database to store its tests.

Since gdbm offered no authentication or access restriction features, we restricted access to our test database by storing it in a particular directory, and then changing the access permissions and ownership of that directory, with the result that only CGI-scripts executed by the local web server could read, write, or execute in that directory. Rudimentary access control to our test database was thus implemented through use of the file system permissions provided by the host machine's operating system.

3.1.2. The test-taker

When one clicks on a link on a web page, the HTML code for the page that subsequently appears may either be downloaded from a file, or it may be generated dynamically by a Common Gateway Interface script (CGI-script), or other modern page-generation tools. Since a CGI-script is a *program*, this mechanism allows an HTML form to provide the *inputs* for a script, which can perform a computation using those inputs, and then generate the HTML to create a web page displaying the result of the computation. Any simple input-compute-output algorithm can be implemented on the web in this fashion.

Our initial design was to use this mechanism as follows: To take a test, a student navigates through a series of pages in which they identify the department offering their course, their course, and finally the test that they wish to take in their course. These steps uniquely identify the test they wish to take.

Once they have selected the test they wish to take, the form shown in figure 2 is displayed, in which students enter their names, student numbers, or alternative means of identification selected by their instructor.

This registration dialogue permits the student's id to be checked against a file of students who are authorized to take the test (in keeping with our security objectives), and provides the identity information needed to report the student's score when they complete the test. In addition to its `Take the test!` button, the form also has a check-box whereby the student can indicate that they are retaking a test they have already taken, for courses in which re-tests are permitted.

Clicking the `Take the test!` button at the bottom of the subsequent page runs a CGI-script that connects to

Figure 2. Logging student ids.

the test database, gets the first question, and generates an HTML form displaying the question. For a multiple-choice question (the only type supported in this prototype system), the form provides *radio-buttons* (a set of clickable buttons of which only one can be selected at a time) for selecting choices, plus a `Submit final answer!` button by which the student commits to a choice and submits the form (containing their answer) for evaluation. A student thus needs no typing skills to answer a multiple-choice question; just the ability to "point and click." Figure 3 presents a sample question, in which the choice `The sun, moon and stars` has been selected.

By using radio-buttons, a student may change their answer at any time prior to the point at which they click the `Submit final answer!` button. This latter button thus provides the mechanism by which a student *confirms their choice*.

The form also supplies two "arrow-buttons" at the bottom of the page: clicking on the right-arrow button skips to the next question; clicking on the left-arrow button allows the student to return to any previously-skipped questions. In addition to their arrow icon, these buttons are color-coded so that a button is green when it can be used to navigate in its given "direction," and red when it cannot be used to navigate. The system thus allows students to answer questions in a non-linear fashion if they so wish.

Clicking the `Submit final answer!` button on the question form activates a CGI-script which processes the form's contents, connects to the test database, checks the student's answer against the information in the test database, and generates a page providing the student with feedback on their answer. This page re-displays the question, annotated with an explanation of why that choice is correct or incorrect, as well as the number of points that choice was worth. Figure 4 presents the feedback page that appears as

Figure 3. A sample question.

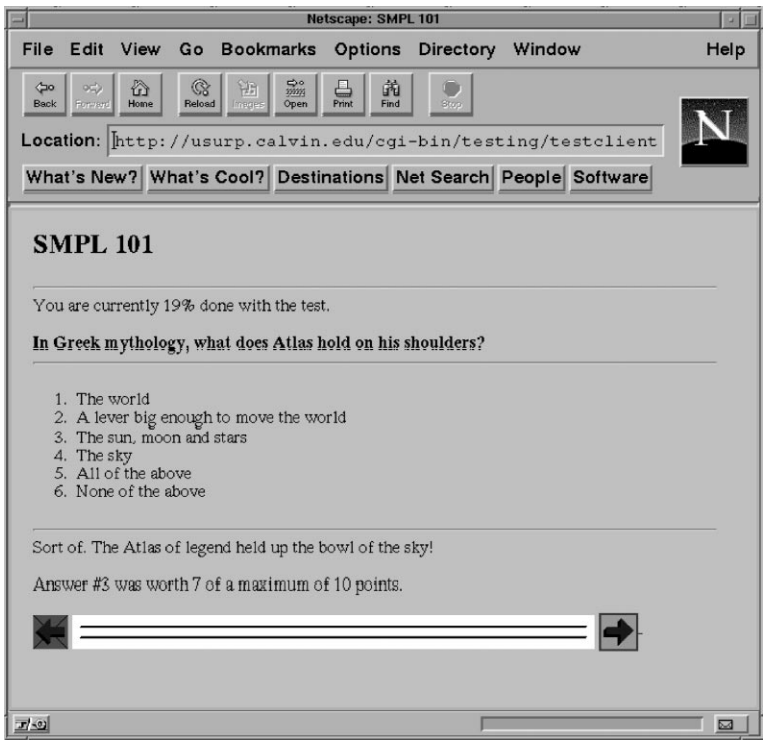

Figure 4. Immediate feedback.

a result of clicking the `Submit final answer!` button in figure 3.

The system thus provides a student with immediate feedback on their response, as well as how many points that response was worth. As figure 4 illustrates, both the feedback and the point value can be customized for each choice, allowing partial credit to be awarded for appropriate choices.

The right-arrow button on this feedback page activates the CGI-script that displays the next question, beginning the cycle over again.

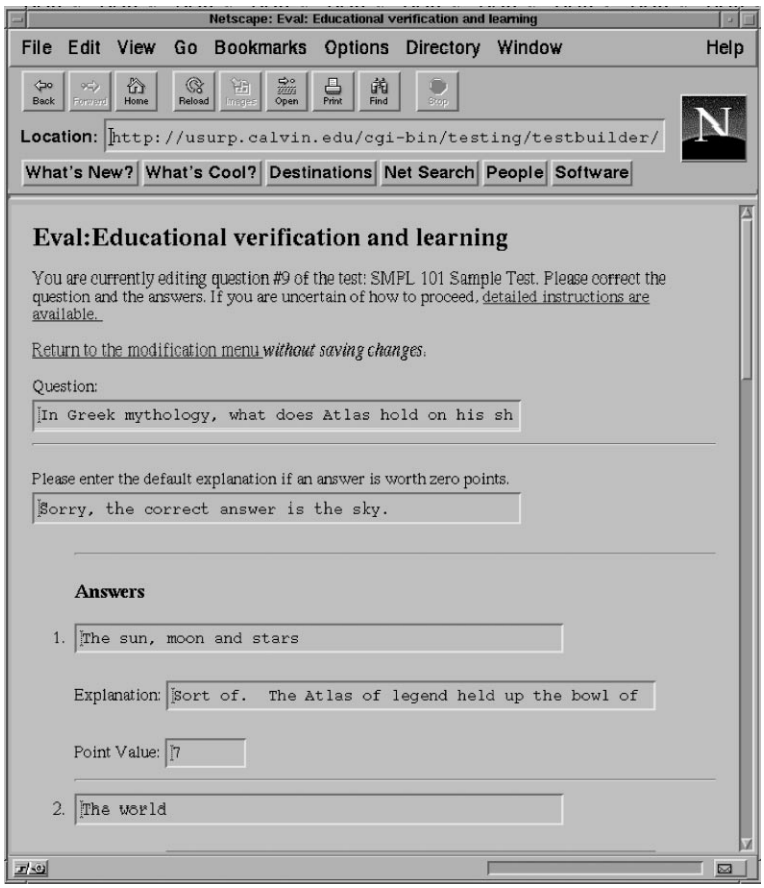

Figure 5. Editing a test (a).

When students complete their test, a summary page is displayed, detailing how many questions were on the test, how many questions they answered, the maximum number of points possible, how many points they earned, and their percentage correct. Students thus leave the testing session knowing precisely how they performed on the test. Moreover, thanks to our immediate feedback mechanism, students can leave the test knowing more about the topic than they did at the beginning of the test session. This allows the process of testing a student to become an integral part of the educational process, rather than just an evil that is necessary in order to evaluate a student's mastery of the material.

3.1.3. The test editor

Our design for the test editor was similar to that of the test taker. An instructor begins by navigating through pages that uniquely identify the department and course for the test being edited. Once the course is identified, the instructor can choose to edit one of the tests listed for that course, or create a new test.

If the instructor chooses to create a new test, a CGI-script generates a page prompting them for a name and password for the test. When the instructor submits a name and password, a second script contacts the test database, creates an entry for a test with this name, records its password, and then displays an empty form in which the user can type a question, the choices to be associated with that question, an explanation for each choice, the point value a choice is worth, a default explanation, and other question-specific information (see figure 5).

Once an instructor has completed a question form, submitting that form activates a CGI-script that stores the question in the test database, after which a blank form for the next question is displayed. The instructor repeats this process for each question in the test.

Choosing to edit a test is similar to creating a test, and even uses the same basic forms. As with taking a test, an instructor begins by identifying the test to be edited by selecting it from a list of links to existing tests. Clicking on a test-link activates a CGI-script that presents a form similar to that of figure 2 asking for the password for that test. Once the password is correctly entered, a CGI-script is activated that contacts the test database, and retrieves and displays a list of links to the questions in the test. Clicking on one of these links activates a second CGI-script that contacts the test-database, retrieves the question whose link was selected, and then displays an editable form containing the data for that question. Figure 5 shows the initial portion of this form.

As shown in figure 5, the form provides text-boxes whereby the user can enter a multiple choice question, as well as the choices to be displayed.

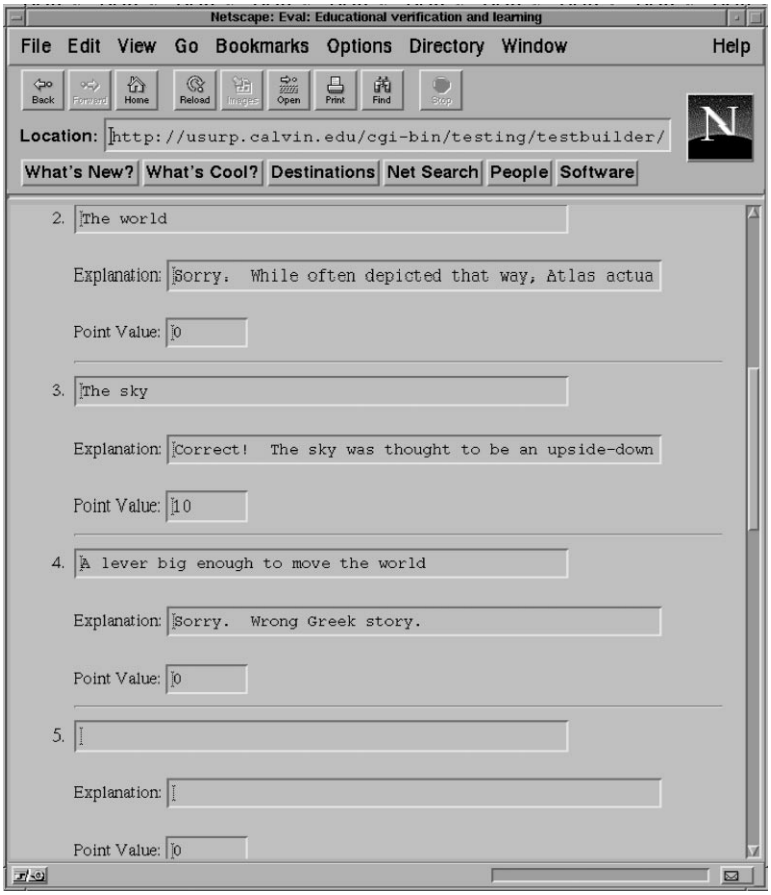

Figure 6. Editing a test (b).

As was discussed previously, one of our objectives is to support *immediate feedback*. To permit this objective to be realized, our form provides a flexible approach:

- For those who prefer to provide one generic explanation for each of the question's choices, we provide a text-box in which a *default explanation* can be entered. This explanation will be displayed for any choice whose explanation text-box (see below) is left empty.
- For those who prefer to associate an explanation with each choice, we also provide an *Explanation* text-box with each choice in which the instructor can write a customized explanation of why that choice is correct or incorrect.

Each choice also has an associated number of points that a student choosing that choice will receive. This permits the test creator to award full, partial or no credit, as appropriate for a particular response.

Figure 5 shows the topmost portion of the editing form, which just shows the question, the default explanation, and information associated with the first choice. The remaining "normal" choices can be uncovered by using the scrollbar to scroll downwards, as shown in figure 6.

As illustrated by figure 6, our system supports up to five "normal" choices, each of which can have its own customized explanation and point value.

Educators sometimes wish to ask questions in which the best answer is some *combination* of choices. To support such questions in a convenient fashion, our form has check-boxes for five "combination" choices, as shown in figure 7.

To include such choices with a question, all a test creator needs to do is click the check-boxes to their left, and they will be automatically appended to the "normal" choices. (In figure 7, the check-boxes for choices `All of the above` and `None of the above` are selected. Since their "Explanation:" text-boxes are blank, a student choosing one of these choices will receive the default explanation.) Of course, these can be ignored, if that is the test-creator's preference. Below these "combination" choices are the remaining options for the question, as shown in figure 8.

The first three text-boxes allow up to three URLs to be specified by which multimedia resources can be incorporated into the question. Following this is a check-box by which the creator of the test can specify that the order of the choices be randomized each time the test is offered.

Last is the `Save Changes!` button: when this button is clicked, an associated CGI-script is activated which processes the information in the form, saving it to the test database, and either

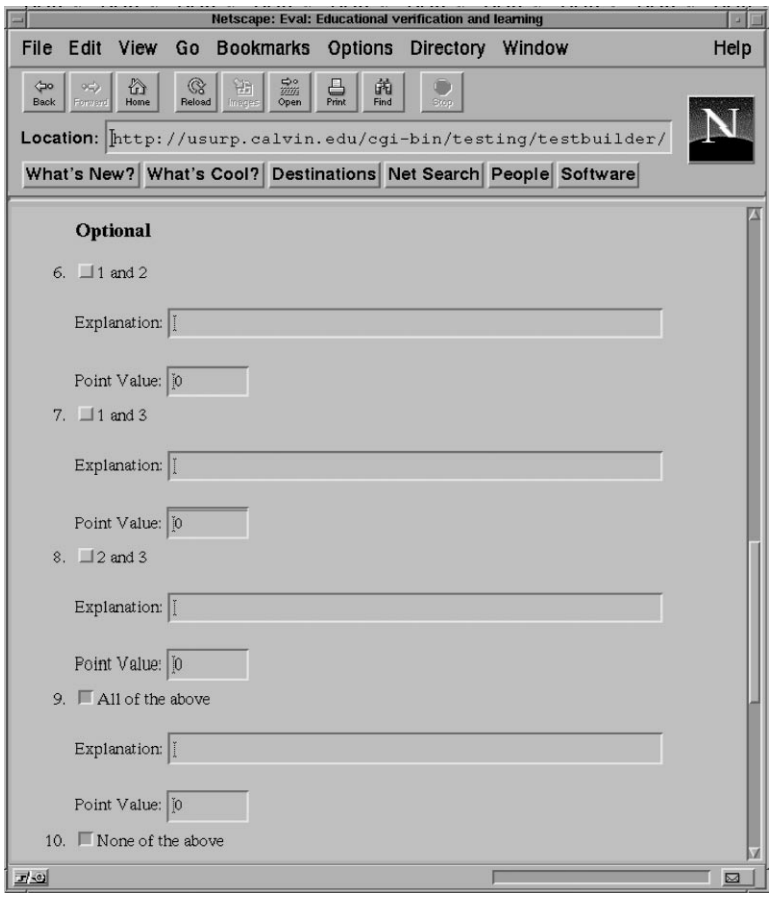

Figure 7. Editing a test (c).

- displays a blank form in which a new question can be entered (if *creating* a test); or
- returns the user to the list of questions in the test (if *editing* a test).

3.1.4. Giving a test

Once a test has been created, an instructor can *give* it. As before, the instructor selects the test to be given by navigating through a series of pages in which he or she specifies the department, course, and finally the test itself. On the last of these pages, the instructor completes a form providing a variety of test-format options (e.g., whether or not questions are ordered randomly, whether retests are allowed or not, the duration of the testing period, and so on). The button to submit this form is thus labeled `Give Test!` and clicking it initiates the testing session.

The submit button on the subsequently appearing page is labeled `End Test!`, and clicking it terminates the testing session. This button activates a CGI-script that gathers the student scores, and computes the average and standard deviation for these scores. The script concludes by logging all of the collected information for this session to a log file, and displaying that information for the instructor.

It is worth noting that if a student takes a test multiple times during the testing period (which can be arbitrarily long), then we use the highest score for computing these statistics, though all scores are recorded. This mechanism can be easily changed to use the first score, or the average score, as desired.

3.2. Refinement 1

Our initial design had a number of drawbacks. In this subsection, we discuss those drawbacks and our first attempts at correcting them.

3.2.1. Storing client-specific information

As we began to implement our initial design, we discovered a significant problem: a WWW-browser is *stateless*, meaning it provides no direct mechanism for storing information returned by a CGI-script, aside from the page being displayed. This is a problem for our initial design, because our system must keep track of a number of client-specific items. For example, for students taking a test, the system must keep track of the student's running score on the test, what questions the student has answered (and thus may no longer answer, even if they use their browser's "Back" button), what questions they have skipped (and thus can still answer), and so on. Such information must be stored somewhere, and since the browser is stateless, we were left with various options:

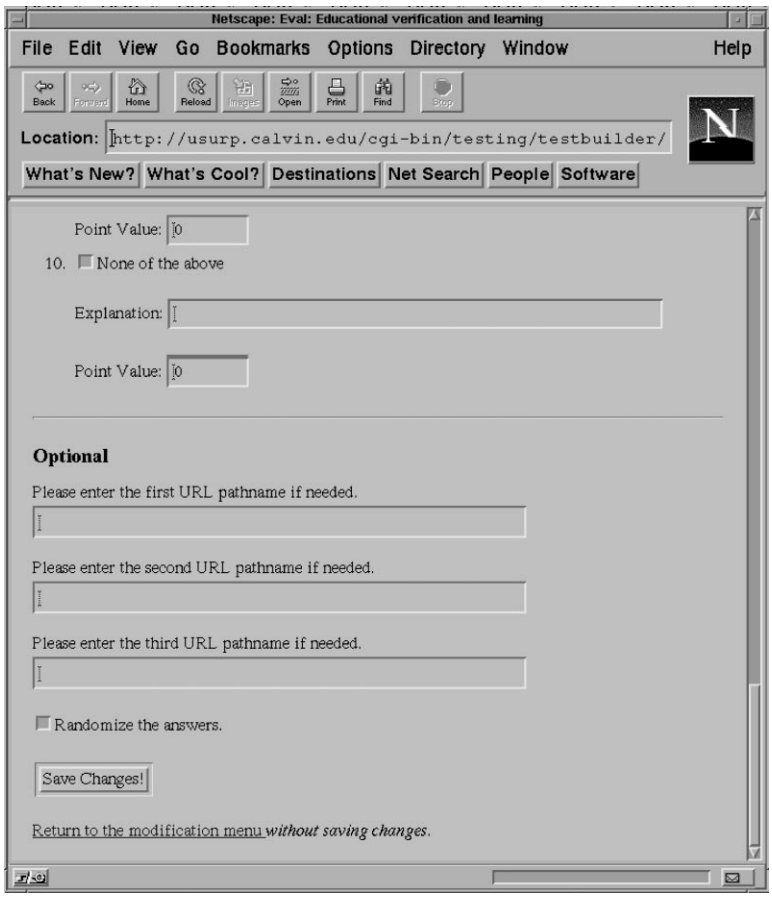

Figure 8. Editing a test (d).

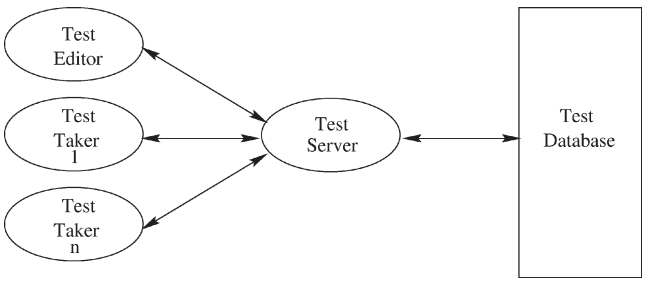

Figure 9. Many-to-one client-server design.

- embed such session-specific information within the page itself, in such a way as to permit the CGI-script to retrieve it;
- write such session-specific information to a file, from which the next invocation of the CGI-script can read the information; or
- add an additional software component to the system to keep track of the session-specific information for a given test.

We chose the last option, and modified our design by adding a *test server* component that is activated when an instructor gives a test. The resulting *client-server design* can be seen in figure 9.

In this design, the server maintains a list in which is recorded the client-specific state information for each active test client. Each time a client takes an action that changes its state, this list is updated, and the update written to a log file, so that the system state can be recovered if the system should crash.

To communicate with the clients, the server uses *sockets*, and listens at a particular port address. Similarly, sockets were added to the client CGI-scripts to enable them to connect to this test-server and send it the result of each interaction, so that the server can update its state information for that client.

For example, when a student answers a question and clicks the `Submit final answer!` button, the client CGI-script activated by that button parses the information in the form to determine which choice the student has selected. It then transmits the student's choice to the test server, and awaits its response.

The test server determines the point value and explanation associated with that choice, updates its session-specific information for that student, and transmits these back to the waiting client CGI-script. Given the explanation and point value, the CGI-script generates the necessary HTML to redisplay the question, followed by the explanation and point value associated with the user's choice, as was seen in figure 4.

3.2.2. Securing the test server

A second drawback of our initial design was that it had CGI-scripts interacting with the test database directly. That is, CGI-scripts are executed by the local web server (typically running as user *nobody* on UNIX systems). This means that even if only the local web server is permitted to access the test database (see section 3.1.1), all an adversary has to do is write a CGI-script that mimics the actions of our client CGI-script, and then start retrieving and/or modifying test information from the database. (Preventing users from writing CGI-scripts on the "official" web server does not close this security hole on multi-user systems, since a user can always install their own web server, configure it to listen on a different port, and run their scripts through it.)

The refined design shown in figure 9 reduces this security hole significantly by preventing the CGI-scripts from contacting the test database directly. Instead, by having the test server act as an intermediary for all database accesses, access to the test database can be restricted to the test server alone.

Just as the most vulnerable point in the defenses of a medieval castle was its gate, the most vulnerable point in our refined design is the test-server, because it provides the entry point to the test database. In order for our system to be secure, access to this test server must be restricted to clients of our system.

Since clients access the test server through a socket, a client must "know" the port number of the test server's socket in order to communicate with it. Thus, one way to make the test server secure is to restrict "knowledge" of the test server's port number to the client CGI-scripts. A first step in keeping this number secure is thus to set the permissions of each client CGI-script so that not everyone is able to read the script.

As a supplemental measure (and to improve performance), we decided to write our client CGI-scripts in a compiled language (C++), rather than an interpreted language (Perl). Whereas an interpreted script consists of the script's *source file* (making it relatively easy to identify the test-server's port number), a compiled CGI-script is a *binary file*, in which locating the test-server's port number is much more difficult (though not impossible).

Since an operating system supports a finite number of ports, a determined adversary could simply "scan" every port in sequence, looking for the one associated with the test server. To counter such a strategy, we devised a special *test protocol*: a special sequence of actions that a client must follow when initiating contact with the test server. Once a client has successfully performed the protocol, the test server will perform whatever transaction it is requesting. When the test server is contacted by a client that does not follow the protocol, the test server records the available information about that client in a special error log and notifies the user giving the test that an unauthorized access was attempted.

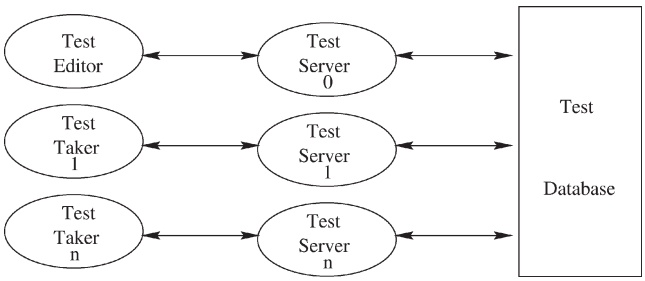

Figure 10. One-to-one client-server design.

So long as knowledge of the details of our test protocol can be restricted to our system's clients and server, our test server (and by extension, our test database) should be reasonably secure. To restrict knowledge of these details to our test server and clients, we rely on using compiled binaries for all of these components. We also took the further step of restricting access to the directory where CGI-scripts were stored, to make it more difficult for an adversary to retrieve and disassemble these binaries.

3.3. Refinement 2

In analyzing our refined design, we became concerned that a single, centralized server could become a communications bottleneck in our system. More precisely, if our server were blocked waiting for a lengthy network or database transaction to complete, and connection requests from clients continued to arrive, then requests could be lost when our server's queue of connection requests becomes filled.

It is well-known that one way to avoid this situation is to employ *multithreading* in the server. By writing a server in such a way that it spawns a new thread of execution to handle every new transaction, a server can continue to handle incoming connection requests at the same time as one or more of its threads are blocked awaiting the completion of a transaction.

However, spawning a new thread of execution is not cheap, and multi-threading should not be used carelessly. That is, a new thread of execution should only be spawned when the amount of work that thread will do is significant, to offset the excess cost of creating the new thread. For this reason, we redesigned the test server to spawn a new thread *the first time it is contacted by a client*. This thread is then dedicated to that client, and remains in place for the duration of the test session for that client. Figure 10 illustrates this refinement.

In our many-to-one client-server design, the (single) test server maintained the session-specific information for its (many) clients in a list. By using a multithreaded server that spawns a separate thread for each client, we achieve a one-to-one relationship between clients and server threads. As a result, a given server thread needs to maintain only the session-specific information for its (one) client. This greatly simplifies the implementation of the server.

3.3.1. The test protocol

Recall from section 3.2.2 that we required clients to follow a prescribed behavior we called the *test protocol* when connecting to a server, in order to be recognized as a valid client. Such a protocol was needed to prevent an adversary from raiding the test database by writing their own unauthorized client.

In its initial form, our test protocol was *static*, meaning it required each client script to behave in exactly the same way to be recognized as a valid client. For example, a possible protocol would be for the client to send the letter E, to which the server would respond with the letter v, to which the client would respond with the letter a, to which the server would respond with the letter l, to which the client would respond with a letter indicating the service the client was requesting of the server. Since the protocol had been followed, the server would perform the requested service and return any results to the client.

Such a *static protocol* has several weaknesses. One of them is *time*, since each interaction between the client and server requires a network transmission, consuming valuable time and bandwidth.

Such a protocol is also weak from a security standpoint: if a determined adversary should discover the prescribed behavior, then they can write their own client that follows that protocol and have full access to the functionality provided by the test server. The weakness of a static protocol is that *all clients perform exactly the same actions*, making it easy to imitate, once discovered.

With the multithreading of our test server, we decided to strengthen our test protocol. The basic idea was to add an element of non-determinism to the protocol, resulting in a *varying protocol* that makes it more difficult for an adversary to imitate a valid client.

In its final form, our test protocol begins when a student registers to take a test (see figure 2), or an instructor enters the password for a test, as described in section 3.1.3. Once this information has been correctly entered and verified, the test server spawns a new thread to handle this particular session. This new thread generates and records a *random password*, passes it back to the client along with the first question of the test, and awaits a response. In our revised test protocol, such random passwords are used as *authentication tokens* for the interactions between server threads and their clients.

More precisely, the client CGI-script that initiated the session receives this randomized password from its server thread, embeds it "invisibly" in the web page it generates (in the form presenting the next question to the user), and terminates. When the user finishes the page and clicks the button that submits its question form, the button-click activates a new CGI-script that extracts the randomized password from the form and returns it to the waiting server thread, along with the rest of the information from the form. The CGI-script then awaits a response from the server.

When it receives a response, the server thread checks the first item it receives against the password it recorded before its last communication. If there is any mismatch, an error is generated and logged to alert the instructor giving the test. Otherwise, the server thread processes the client information that follows the password and performs whatever service(s) it requests (e.g., add 10 to the cumulative score, get the next question, etc.). The server thread then generates and records a new randomized password, and sends this new password back to the waiting client with the requested information, beginning the cycle anew.

Each interaction between the client and its server thread thus involves a *different* randomized password, generated by the server and sent to the client, that the client must subsequently return to the server as an authentication token for its next transaction.

To prevent a determined adversary from intercepting the randomized password and/or test information (e.g., using network "sniffer" software), we also used public key encryption at both the client and server ends to encode the information to be sent across the network. At the time we began building our prototype, the CGI mechanism did not provide a built-in encryption mechanism. As a result, we implemented our own set of C++ secure-socket classes to ensure that client-server communication remained private. Since that time, standardized security mechanisms have appeared, including the **Secure Socket Layer** (SSL) and **Secure-HTTP** (shttp), which provide more convenient means of achieving secure client-server communication [Tittel *et al.* 1996].

The resulting protocol eliminates the time weakness of our static protocol, and significantly reduces its security weaknesses. However, even this revised test protocol is not completely secure. In the next section, we discuss some of its remaining insecurities.

4. Security issues

Trying to build a secure system is the modern-day equivalent of trying to build a medieval castle capable of withstanding a siege. The system's "entry points" create vulnerabilities, and so must be fortified and guarded against determined adversaries. Its "walls" must be built upon a foundation that will foil attempts at "undermining" them. Its owners must be alert to attacks, ready to repel invaders, and informed about the "ladders" and "siege engines" adversaries will use to try and break its defenses.

Throughout section 3, we discussed the evolution of our prototype's design in response to various security and performance issues. We can summarize our security measures as follows:

- To guard against unauthorized attempts to read an existing test, tests are password protected. A test's password serves as an authentication token for the first contact between a test editor client and the test server.
- To guard against students having a surrogate take their test for them, a student must register for a test by spec-

ifying some name, which can be specified by the instructor, and can be different for every test. The name by which a student registers serves as an authentication token for the first contact between a test taker client and the test server, which generates a new server thread dedicated to that client.

- To guard against connections from unauthorized clients, a server thread generates a new random password each time it is contacted by its (authorized) client. The server thread transmits this password to its client when it returns whatever information the client was requesting. In its next interaction, the client must return this random password back to its server thread in order to receive service. The random password thus serves as a client's authentication token for each contact after the initial contact.
- To guard against theft of the password (or test information) in transit across the network, all transmissions are encrypted using public key encryption, with clients and servers having distinct private keys.

While it might seem as though these precautions would make our prototype reasonably secure, this is unfortunately not the case.

4.1. The distance learning testing problem

One insurmountable problem is the distance learning environment itself, where the students are in an uncontrolled environment: it is quite easy for a person taking a test in such an environment to cheat by consulting some outside authority (a text book, their spouse, etc.) for answers as they take the test. In such a situation, the completed test reflects the knowledge of that outside authority, rather than the knowledge of the person who supposedly took the test.

Since this problem seems to stem from the nature of the distance learning environment, we call it the *distance learning testing problem*, and we know of no perfect solution for it. While video telecommunications could be used to monitor distance learners as they take a test, the invasiveness of such a solution seems Draconian, smacking of "Big Brother" peering into one's home.

From our point of view, a preferable, if imperfect solution is to employ an *honor code*, in which at the conclusion of a test, test takers affirm on their personal honor that the test represents their own work and not that of anyone else. Such a system seems to be consistent with the spirit of distance learning: By assuming that most students are honest, it promotes an atmosphere of trust, rather than an atmosphere of suspicion. Dishonorable people who abuse such a system may reap some short-term gains, but will eventually get their due when *The Peter Principle* [Peter and Hull 1969] catches up to them.

4.2. The Collaborator Problem

It might seem that if we were to ignore the distance learning environment, and construct a special *test facility* in which students show a photo-id to be admitted and take their on-line tests under the supervision of a proctor, that our prototype would be secure against cheaters. This is unfortunately not the case.

While the above precautions *do* make the system relatively secure against *solo* outsiders trying to break into the system, they *do not* make the system secure against outsiders seeking to gain entry *with the aid of someone already in the system*. Just as the strongest medieval castle was vulnerable to a traitor from within opening its gate to invaders, so our system's security can be compromised if a collaborator from within "opens its gate" to an outsider.

To illustrate, suppose that there are two roommates, Joel and Aaron (no relation to the authors). Joel is a senior English major taking the freshman Computer Science course as an elective, while Aaron is a senior Computer Science major. Suppose further that Joel proposes to Aaron that Aaron take his Computer Science exam for him, and that Aaron (relishing the very idea of beating the system) agrees to do so. In spite of our efforts to create a secure system, it is quite easy for the roommates to breach the security of our prototype, even if the test is administered in a special test facility as described above. Breaching the security is easy because of the nature of the interaction between a web browser and a CGI-script.

In our test protocol, each client CGI-script needs the random password generated by its server thread during the *previous* interaction, since that password serves as the authentication token for its next interaction with its server thread. Since the client was not active when the random password was generated, and the browser itself does not provide any means of storing such information, the client that receives the random password must store it somewhere where the client that will use that password can retrieve it.

One place to do so is in the page itself, within the URL by which the CGI-script will be invoked, as an "argument" to the script.[1] The problem is that when we store the authentication token somewhere within the page, we effectively make it publicly available to the person viewing that page, and to anyone with whom they can communicate.

To illustrate, all that Joel has to do in order for Aaron to become his electronic surrogate is:

(1) register for the test in the normal manner;

(2) answer its first question to the best of his ability, and wait for the system to display the explanation associated with that choice;

(3) select the `View -> Document Source` menu choice on the browser;

(4) identify the URL to the next question. This is not difficult, since it is the URL associated with the right-arrow button (see figure 4);

[1] Another place is within a "hidden" field within the form; another place is in an external temporary file. All of these approaches share similar vulnerabilities to attack.

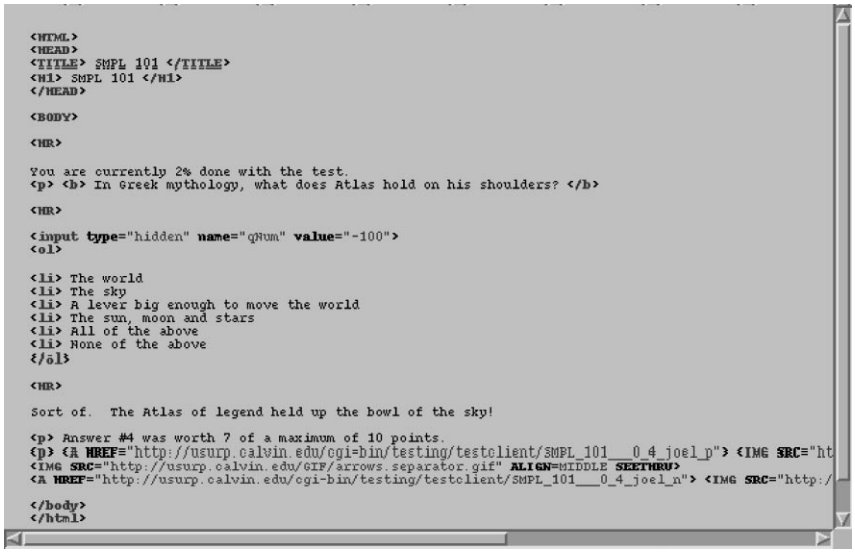

Figure 11. `View -> Document Source` under Netscape.

(5) give that URL to Aaron (through an e-mail message, a trip to the rest room, or some other means); and

(6) act busy throughout the rest of the test.[2]

Figure 11 shows a portion of what is displayed when a person using the test-taker client (see figure 3) selects the `View -> Document Source` menu choice under Netscape. It is quite easy to pick out the URL associated with the right arrow, near the bottom of the figure:

While Joel pretends to complete his test, Aaron can copy the URL into his browser and take the test as an *electronic surrogate* for Joel, circumventing our security measures.

Note that the only knowledge Joel needs is the ability to identify a URL. He need not understand where the authentication token resides within the URL, or anything else about its meaning, other than where it begins and where it ends. This means that all he needs is a rudimentary knowledge of HTML, which is easily acquired. Note also that once the URL has been given to Aaron, Aaron can take the test from anywhere on the Internet!

Note finally that Joel can pass whatever elaborate identity checking procedure we require to be admitted to our "controlled" testing facility, such as presentation of a photo-id. It is interesting that unlike a traditional cheater, Joel does not beat the system by *receiving* information; but does so by *giving away* information (i.e., the URL to the second question) to Aaron.

Because of the variety of ways this URL could be given away, preventing this from occurring requires a testing facility that is hermetically sealed (both physically and electronically) from the outside world. That is, it is not sufficient to simply disable the e-mail capabilities of the test facility's machines, if Joel can simply rendezvous with Aaron in the rest room, or walk past him in the hallway en route to the rest room.

For lack of a better name, we call this problem the *Collaborator Problem*. Because the problem stems from our use of multiple, CGI-generated forms (one per question) that must be accessible from anywhere on the Internet, any multi-form, CGI-based application for the Internet must take special precautions to guard against this problem.

4.2.1. Solving the Collaborator Problem

There are a variety of approaches to solving the Collaborator Problem. Some of the ones we have explored are as follows:

Restricting access. If a web server is configured appropriately, a file can be placed in a directory that prevents machines outside of a specified set of IP addresses from accessing that directory and its subdirectories. By storing a test in such a restricted-access directory, and restricting access to the machines in a controlled testing facility, this mechanism can be used to prevent Aaron from accessing the test from outside of our testing facility. Since this is the equivalent of "hermetically sealing" the testing facility from electronic contact, it can be used to solve the problem.

This approach requires that the IP addresses of the machines used to take the test be known in advance. This may present an administrative problem at sites where students bring a notebook computer and plug into the network. Other administrative difficulties must be resolved if the same test can be accessed from different test facilities at different times.

Using web-server log files. When a browser requests a URL from a web server, that server records in a log file the requested URL, plus the IP address of the client machine making the request (or its name). This presents an alternative solution to the problem that is somewhat

[2] Alternatively, in situations where the feedback is being delayed to the end of the test, Joel could (i) select the `View -> Document Source` menu choice on the first question, effectively sacrificing the first question; (ii) identify the appropriate URL from its `<FORM ACTION ...>` tag; (iii) give the URL to Aaron; and (iv) act busy the rest of the test.

more flexible: initially, the test directory contains a file that gives no one access to the test. When an instructor gives a test (i.e., starts its server), the CGI-script starts the test server and then waits momentarily while the students register for the test. After this pause, it scans the log file for entries listing accesses to the URL of the Registration form, and records the IP addresses in those entries. By doing so, it identifies the IP addresses of the machines in the testing facility, be they plug-in notebooks or otherwise. It then *dynamically* builds an access-restriction file that grants access only to those machines, and drops that file into the test directory. Since Aaron cannot go through the Registration dialogue (because the instructor can specify the name under which a student registers for a test), his IP address will be prevented from accessing the test directory, solving the problem.

A significant drawback to this approach is the problem of tuning the time-delay between when a student can access the Registration dialogue and when they can access the first question. The delay must be sufficiently long for *all* students to access the registration dialogue, in order to get entered in the log file. However, the longer the delay, the greater the number of students who will complete their registration before the access-restriction file is put into place, and whose access to the first question will be delayed. While they can simply click on the "Reload" button on their browser until they receive access, a system that requires this of its users is inelegant, and increasing the stress of hyper-caffeinated students at the beginning of a test in this manner seems unkind.

Authentication cookies. An alternative approach is to have our system's web server deposit a *cookie* [Netscape 1997] on the client's system during the Registration procedure, and then check for the presence of this cookie during each subsequent interaction, denying access if the cookie were not present. The cookie thus serves as a supplemental authentication token to our randomized passwords. (Alternatively, the randomized password can be stored in the cookie.) In our scenario, the cookie would be deposited on Joel's machine but not on Aaron's, effectively locking Aaron out of the system.

One drawback to this approach is that many users distrust cookies and disable the mechanism on their system (e.g., by making the cookie file read-only). Such users must be convinced to trust the system in order for this approach to succeed. However this should not be a problem in a controlled testing facility, unless students are bringing their own notebook machines, in which case they must be persuaded to trust the system.

A bigger drawback is that cookies are stored within files which, when opened, provide a user with access to the cookie. As was the case with randomized passwords, cookie-based security can be breached if a person with the cookie extracts it from its file and gets it to their roommate (along with the URL to the second question), who plants it in their cookie file and proceeds to take the rest of the test as an electronic surrogate.

Both of these drawbacks can be circumvented by using encrypted "fast-expire" cookies (i.e., cookies with no `expires` attribute). Such cookies are not stored in files, and are discarded at the end of the browser session. By storing within the cookie the user's name, randomized password, IP address, time, question number, and other session-specific information, and changing the cookie for each question, our test protocol can effectively be implemented using cookies. By encrypting this information, it can be rendered effectively inaccessible to collaborators like Joel and Aaron.

Using environment variables. CGI-scripts have available to them a variety of *environment variables*, whose values they can inspect. One of these is the `REMOTE_ADDR` variable, in which the IP address of the machine that invoked the CGI-script is automatically recorded [Tittel *et al.* 1996]. This provides an alternative mechanism for foiling collaborators: each time it is invoked, the client script can retrieve the IP address of its invoker, and pass this on to its test server. During its first interaction with its client, the server thread records this IP address. During each subsequent interaction with its client, the server thread compares its recorded IP address against the one sent by the client in the current interaction. If there is a mismatch, the server thread can "raise the alarm" and take an appropriate action.

Using Java. Another solution is to rebuild the client side of the system, by replacing our client CGI-scripts with Java applets. Unlike a CGI-script, the "lifetime" of a Java test-taking (or editing) applet will be the duration of the test session. Since the applet will persist between interactions with the server thread, all session-specific information (including the information of an authentication token) can be maintained internally within the applet, rather than embedding such information publicly in a web page.

Such a system can also be built either as a stand-alone Java *application*, or as a web-based Java *applet*. We believe that each should be equally secure – a hypothesis we are currently examining.

While it is possible through the use of cookies and/or environment variables to solve the Collaborator Problem, we believe that Java provides the best overall solution to the problem. In the next section, we discuss our future plans in that regard.

5. Conclusions

We have examined the problem of building a secure online testing system that uses the World Wide Web as a testing medium. To explore the problem, we have built a prototype system using common gateway interface (CGI) scripts. Through this prototype, we have learned that the openness of the CGI mechanism can make it difficult to

build a secure, multi-form Internet application. Achieving security requires that extra precautions be taken, and it is difficult to anticipate every possible source of attack.

5.1. The future

As a result of our experience in building our prototype, we have decided to apply object-oriented analysis and design [Booch 1994] to redesign our system as an object hierarchy. By applying object-oriented design to the problem, we can design *an extensible system* – one to which new kinds of questions and new kinds of tests can be added without modifying any of the existing code. For example, a `Question` will be an *interface* in this system, which objects like `MultipleChoiceQuestion`, `CompletionQuestion` and `MatchingQuestion` must implement. If we subsequently decide to add `EssayQuestion` objects, none of the existing code will have to be changed, provided the `EssayQuestion` class implements the `Question` interface. Similarly, a `Test` will be an interface that our `LinearTest` class will implement. If we subsequently wish to add `AdaptiveTest` objects to our system, none of the existing code will require modification, provided `AdaptiveTest` implements `Test`.

We have decided to implement this system using Java for a variety of reasons, including

- Java applets and applications can be run on each of the major platforms, enabling us to achieve our multiplatform objectives.
- Java provides built-in, standardized sockets supporting thread communication anywhere on the Internet, allowing us to achieve our distance-learning objectives.
- Java provides built-in, standardized graphics capabilities, allowing us to achieve our GUI objectives. These will allow us to define convenient point-and-click GUIs for a rich variety of question types, including multiple choice, completion, true/false, matching, and essay questions.
- Java applets and applications can access, retrieve and play on-line multimedia resources via a URL, allowing us to achieve our multimedia objectives.
- Java makes it easy for us to provide different versions of the system: a stand-alone *application* version for those who want to use the system on an intranet, and a web-based *applet* version for those who would prefer an open-to-the-world web-based system. Preliminary investigations indicate that both types of systems can achieve our security objectives.
- Java provides built-in, standardized multithreading which will be useful in writing our multithreaded test server.
- Java provides the Java Database Connectivity (JDBC) mechanism. JDBC allows us to build our system in a manner that is independent of any particular database package, but can interface with nearly any shareware or commercial database package. See `http://www.javasoft.com/jdbc` for a complete list of the database packages for which support is available.

We plan to call this new system **pita** for **p**latform **i**ndependent **t**esting **a**pp. We have completed its object-oriented design, and are currently working at implementing that design. Details of the pita system will be the subject of a future report.

References

Asymetrix (1996), *Toolbook CBT*, Asymetrix Corp. `http://www.asymetrix.com/`

Berners-Lee, T., J. Cailliau, J. Groff, and B. Pollerman (1992), "World-Wide Web: The Information Universe," *Electronic Networking: Research, Applications and Policy 2*, 1, 52–58.

Booch, G. (1994), *Object-Oriented Analysis and Design*, Benjamin Cummings, Redwood City, CA.

Derringer, D. (1985), *Technology Advances that May Change Test Design for the Future*, Educational Testing Service, pp. 35–43.

Garfinkel, S. and G. Spafford (1996), *Practical UNIX and Internet Security*, O'Reilly and Associates, Sebastopol, CA.

GNU (1997), *GNU Database Manager*, Free Software Foundation. `http://www.gnu.org/`

Harker, G. (1956), "Delay of Reward and Performance in Instrumental Response," *Journal of Experimental Psychology 51*, 303–310.

Icove, D., K. Seeger, and W. VonStorch (1995), *Computer Crime*, O'Reilly and Associates, Sebastopol, CA.

Johnson, T. (1997), "How to Install QUIZIT." `http://video.cs.vt.edu:90/ telfer/install.html`

Leclercq, D., E. Boxus, P. de Brogniez, and F. Lambert (1993), *The TASTE Approach: General Implicit Solutions in Multiple Choice Questions, Open Book Exams, and Interactive Testing*, Springer-Verlag, pp. 210–232.

LXR*Test (1994), *LXR*Test*, Logic eXtension Resources. `http://www.lxrtest.com/`

Macromedia (1996), *Authorware CBT*, Macromedia Inc. `http://www.macromedia.com/`

Marker, G. and L. Ehman (1990), *Linking Teachers to the World of Technology*, Educational Technology Publications, pp. 22–26.

Miller, D. (1998), "Medical Council of Canada." `http://www.mcc.ca/mcc_ex.htm`

Netscape (1997), *HTTP Cookie Specification*, Netscape Communications Corp. `http://www.netscape.com/newsref/std/cookie_spec.html`

Perin, C. (1943), "A Quantitative Investigation of the Delay-of-Reinforcement Gradient," *Journal of Experimental Psychology 32*, 37–51.

Peter, L. and R. Hull (1969), *The Peter Principle*, W. Morrow, New York.

Question Mark (1995), *QM Web*, Question Mark Computing Ltd. `http://www.questionmark.com/`

Raval, R. (1993), "Closing the Distance Between Student, Teacher and Parent," *Technological Horizons in Eduction 9*, 75–78.

Russell, D. and G. Gangemi (1991), *Computer Security Basics*, O'Reilly and Associates, Sebastopol, CA.

Tinoco, L., E. Fox, R. Ehrich, and H. Fuks (1996), "QUIZIT: An Interactive Quiz System for WWW-Based Instruction," In *Proceedings of the VII Brazilian Symposium of Informatics in Education*, Belo Horizonte, Minas Gerais, pp. 365–378.

Tinoco, L., E. Fox, and N. Barnette (1997), "Online Evaluation in WWW-based Courseware," In *The Proceedings of the Twenty-eighth SIGCSE*

Technical Symposium on Computer Science Education, C. White, Ed., Volume 1, Association for Computing Machinery, San Jose, CA, pp. 194–198.

Tittel, E., M. Gaither, S. Hassinger, and M. Erwin (1996), *CGI Bible*, IDB Books, Foster City, CA.

Vockell, E. and J. Hall (1989), "Computerized Test Construction," *The Social Studies 80*, 114–121.

Wainer, H. (1990), *Computerized Adaptive Testing: A Primer*, Lawrence Erlbaum, Mahway, NJ.

WebCT (1997), *WebCT*, University of British Columbia. `http://homebrew.cs.ubc.ca/webct/`

Winn, B., B. Ellis, E. Plattor, L. Sinkey, and G. Potter (1990), *The Design and Application of a Distance Education System using Teleconferencing and Computer Graphics*, Educational Technology Publications, pp. 76–80.