Courses as Code: The Aquinas Learning System

W. Michael Petullo
mike@flyn.org
University of Wisconsin–La Crosse
La Crosse, Wisconsin, USA

ABSTRACT

Aquinas aims to maximally apply an everything-as-code approach to teaching the practice of programming and exploit development using hands-on exercises. Teachers define exercises using a machine-readable format, and Aquinas processes these definitions to setup artifacts such as instructions, grading scripts, and network targets. Students submit solutions using Git and benefit from immediate grading. This paper describes the design and implementation of Aquinas, and it evaluates Aquinas's effectiveness as a teaching tool in undergraduate and graduate courses. Aquinas is open source software.

CCS CONCEPTS

• **Social and professional topics → Computer science education**.

KEYWORDS

computer science education, computer security, exercises

ACM Reference Format:
W. Michael Petullo. 2022. Courses as Code: The Aquinas Learning System. In *Cyber Security Experimentation and Test Workshop (CSET 2022), August 8, 2022, Virtual, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3546096.3546099

1 INTRODUCTION

Imagine Tom, a teacher who provides instruction in topics related to programming and computer security. Tom wants to provide high-quality practical exercises for his students, but the development and maintenance of these exercises consumes too much of his time. He often finds that changes to his environment break his exercises, and a proliferation of virtual machine images has left him overwhelmed. He finds himself fighting against course-management software that expects he use tools not developed for the fields of computer science and software engineering.

Imagine Samantha, a computer science student who loses herself in tools unrelated to her future profession. She submits her assignments using email, or she manually uploads them to a course-management software service. Everything seems chaotic, and she loses track of who has what version. She does not yet realize that better tools exist for managing changes and group work. Her professor mentions such tools, but she is left with little time to learn more. She will be surprised when she arrives at he first job to find a team expertly managing its operations using Git.

We know that experiential learning enhances the education of students in the field of computer science [8, 14]. Furthermore, Capture-The-Flag exercises (CTFs) and other competitive events blend the value of experiential learning with the incentives of games. Such exercises have shown a great deal of value, especially when learning low-level systems programming and security [10, 22, 28]. Yet building and maintaining these exercises remains hard.

We wondered if an answer could be found in modern software engineering practices, after observing people who had expanded these practices beyond their original application. Infrastructure-as-code [15] applies proven software development techniques, such as revision control, continuous integration, and continuous delivery, to network and system administration. An even broader application of this approach is sometimes referred to as everything-as-code.

This paper describes Aquinas, an interactive learning system inspired by the everything-as-code approach along with ideas for improving the convenience and repeatability of the CTF format. Aquinas builds on other efforts to apply everything-as-code to the process of educating the next generation of programmers and software-security experts.

Aquinas was also inspired by the resurgence of what we now refer to as free and open-source software [20], which occurred during the last few decades. Hosted Git providers now ease the dynamic of proposing, testing, and accepting modifications to open-source software. These developments integrate well into all levels of education, and they could enhance and broaden education in the practice of programming with an eye toward computer and network security.

The overall objective of Aquinas is to develop and study an interactive learning system that applies everything-as-code techniques, integrates well with existing and emerging hosted infrastructure, and aids teachers who want to develop, distribute, and maintain a wide range of programming exercises. Of particular interest are the type of low-level, security-minded exercises found in CTFs, which have until now required meticulous development and maintenance.

Since Aquinas is open-source, other educators can adopt its use, and we expect that it will be light enough to ultimately permit hosting the system for third parties at little cost using a software-as-a-service model. The client-side requirements of interacting with Aquinas will remain limited and freely available, thus allowing the use of Aquinas beyond university education. We would like to apply Aquinas to middle- and high-school curricula or industry training.

This paper begins with a survey of related work. §3 describes the motivating factors behind the design of Aquinas, §4 describes

CSET 2022, August 8, 2022, Virtual, CA, USA

ACM ISBN 978-1-4503-9684-4/22/08…$15.00
https://doi.org/10.1145/3546096.3546099

Aquinas's design, §5 evaluates Aquinas along a number of dimensions, and §7 lays out future work surrounding Aquinas. Three parties appear in the later sections: *Samantha*, a student; *Tom*, a teacher; and *Alex*, a system administrator. The paper's conclusion summarizes what we learned while using Aquinas to teach.

2 RELATED WORK

An illustrative example of the application of everything-as-code to course materials was the work of Rodriguez, Allison, Apsey, and Boudreau [21]. Their work set the foundation for the U.S. Army's Cyber School curricula, and their approach achieved a remarkable scale and pace of collaboration. They focused on managing lectures, assignments, exercises, infrastructure, and other related items using a distributed version control system and software-development practices.

Clifton et al. managed a fundamentals course sequence using a version control system [6]. Their work required students in an introductory computer science to make use of Git. Git allowed teams to work in parallel, students to continuously commit progress, and students to pull feedback from the same repository that hosted their submissions. Instructors were also able to work in parallel, and they found a number of surprising benefits to using Git. We found hope in the results they had using Git with students who were new to programming.

A number of approaches have made experiential learning more repeatable. SEED [8] built its labs using virtualization and either Linux or Minix, both freely available. Codecademy [23] provides browser-based lessons and exercises in fields related to computer science and programming. NoobLab combines instruction, a programming test bed, and assessments [14]. Webseclab provided an environment for exercises related to teaching web application security [2]. Indeed there exist many options for online learning in our field.

Guo studied how systems could replicate the best of classroom environments in freely-accessible, online settings. His work created the Python Tutor [11] online programming and program visualization platform; Codechella [13], which adds collaborative features to Python Tutor; and Codeopticon [12], which adds features to allow a single tutor to help multiple learners. Each of these improvements made it easier for students to receive high-quality tutoring while completing programming exercises online.

Capture-the-flag exercises have blended the value of experiential learning with the incentives of games, and they have emerged as a popular means of providing low-level systems programming and security exercises. The gamification of computer science education has inspired a number of workshops, including the USENIX Summit on Gaming, Games, and Gamification in Security Education and the USENIX Workshop on Advances in Security Education. The CTF format comes in a number of flavors, most commonly Jeopardy-style, variations of attack-defend, and red-team exercises.

Jeopardy-style CTFs challenge competitors with a series of problems that the competitors select and complete in, following an order of their choosing. These CTFs categorize their problems, similar to the game show that inspired the name of the CTF class. One example is CyberStakes, most recently sponsored by the US Army. Carlisle et al. provide a description of integrating CyberStakes into an undergraduate curriculum [3].

Cowen et al. described the CTF competition at DEF CON 2003 [7]; DEF CON's CTF is one of the longest-running and most renowned of the attack-defend variety. A number of efforts aim to turn the attack-defend CTF into a pedagogical tool for teaching system security. Researchers from MIT Lincoln Laboratory ran a competition preceded by a series of security lectures [28]. Andrew Ruef et al. proposed a style of competition they named "build it, break it, fix it" [22]. In build it, break it, fix it, competitors first implement a software system according to a given specification. Next, referees grade these submissions before allowing the other teams to find defects in them. In the final phase, competitors fix the defects other teams found in their work.

The US National Security Agency (NSA)-sponsored Cyber Defense Exercise (CDX) [10] is an example of a red-team exercise, so named because a red team aligned with the competition referees serves as the sole attacker. These exercises task competitors with mimicking large-scale networks that they then defend. Red-team competitions allow entire courses to surround learning, building, and defending a network.

The popularity of the CTF format has led some to release their underlying software under open-source licenses. CTFd follows from the work to administer CSAW, and its developers also provide managed deployments [4]. picoCTF comes from Carnegie Mellon University, was originally targeted at applying the CTF format at the high-shool level, is the system behind the eponymous competition, and went on to serve as the infrastructure behind a number of other competitions. PicoCTF tries to lower the bar for use by allowing challenges to be completed entirely from a web browser. iCTF is a framework geared towards attack-defend exercises [27].

Chung and Cohen describe some of the pitfalls present in the CTF format [5]; these include the barrier to entry, the difficulty in designing pedagogically-sound challenges, quality assurance, the impact of assigning point values, and the exercise infrastructure.

3 MOTIVATION

Aquinas presently exists as a research prototype, available under the GNU Affero General Public License. The following goals drove Aquinas's design:

(1) maximally employ everything-as-code;
(2) allow for exercises that involve network programming and exploit development while protecting student work and course infrastructure;
(3) facilitate easy-to-define exercises with a consistent specification language;
(4) ease reuse of exercises across many programming languages;
(5) allow for high-quality assignment instructions;
(6) provide a web- and Git-based interface to students that mirrors industry practices;
(7) provide for automated grading and student feedback; and
(8) apply the principle of least privilege and use a type-safe language to protect student work and infrastructure.

SEED inspired Aquinas, but Aquinas additionally eases the task of writing new exercises and delivering them. We describe the syntax and facilities Aquinas provides for this below. Consistent with

everything-as-code, Aquinas stores lectures, assignments, student submissions, grading results, and more in a revision control system.

We wanted the operation of Aquinas by Tom, our imaginary teacher, to appear very much like any other software development endeavor. Our design allows Tom to manage all of his course materials using Git, and Tom benefits from Aquinas automatically processing many of the artifacts he creates. Likewise, we set out to benefit Samantha, our imaginary student, by guiding her to use modern software-development tools. The idea is that interacting with Aquinas makes use of these tools, encouraging practice and thus reinforcing learning. Aquinas purposely deviates from those learning-management systems that are modeled on business applications such as word processors or slide-show editors.

Aquinas supports a wide range of exercises, but it is particularly well suited for the challenges found in the Jeopardy-style CTFs. Aquinas adjusts the CTF format for use in a curriculum, rather than favoring a strictly self-taught approach. Yet Aquinas does allow students to enrich their experience through self teaching, and it sometimes points out how a student could further investigate a topic. The design of Aquinas addresses Chung and Cohen's concern about quality assurance; this follows from our own experience with keeping fragile exploit-style homework assignments fresh.

4 DESIGN AND IMPLEMENTATION

Aquinas first served as an educational aid within the US Army's Cyber Solutions Development Detachment (CSD-D). Aquinas was one tool we used in our educational program, which provided curricula that spanned 12–18 months. We taught non-programmers from diverse backgrounds who demonstrated an aptitude for programming the skills necessary to be contributing members of a programming team. In total we certified roughly 75 candidates over a three-year period. The author has subsequently used Aquinas as a core part of undergraduate and graduate curricula at University of Wisconsin–La Crosse.

Aquinas presently provides 292 lessons spanning six programming languages, namely AMD64 Assembly, Bourne Shell, C, Go, Python, and Java. Lessons range from introducing students to Unix, a developer's environment, and Git to asking the student to exploit a vulnerable program through the use of Return-Oriented Programming (ROP) [1, 24] or tailor an SELinux [25] policy to support custom software.

The source code and documentation for Aquinas is available at https://www.flyn.org/projects/Aquinas, and the canonical instance of Aquinas is at https://www.aquinas.dev/. (Only select email addresses can presently self-register.) Work on Aquinas began in September of 2018.

The rest of this section will imagine a scenario where our teacher Tom aims to deploy a lesson on ROP to his students, including Samantha. Tom wants his students to write a program that uses ROP to exploit a vulnerable network service, extract a secret string, and print that string to standard output.

Tom the teacher's view

Tom defines exercises for two audiences: human students like Samantha and Aquinas itself. Writing for students is a matter of specifying a lesson and exercise description using LaTeX—or, alternatively, Markdown—and pushing that description using Git. Aquinas provides LaTeX commands to make this easier, including conditionals that allow Tom to specify several language-specific exercises using one source document. Aquinas transforms Tom's LaTeX into the HTML that it ultimately delivers to student browsers.

Writing for the second audience—Aquinas—consists of using a syntax defined by Aquinas to describe exercises so that Aquinas can deploy and grade them. Tom wants to write an exercise that requires students to use ROP to exploit a network service in a way that causes the service to reveal a secret; the ability of a student solution to print the secret indicates success.

An exercise like this is difficult to define and maintain without Aquinas. Tom would normally have to take care to compile the target program in a way that does not break the target's vulnerability. Tom might also want to provide students with a redacted copy of the binary to allow for analysis and testing. Additionally, Tom must set up the environment surrounding the network service, along with firewall rules and other control measures. Furthermore, the artifacts involved in the exercise are fragile and changes in the surrounding environment (e.g., updates to the compiler, shared library, and host system) can cause them to break. Thankfully, with Aquinas Tom can define his exercise using a JSON-based syntax that allows Aquinas to automate much of this tedious setup work.

Figure 1 lists the definition of Tom's "rop" exercise. Tom's defintion includes a summary along with the languages the exercise allows. Aquinas will create a variant of the exercise's page for each language. Tom's use of `tags` allows Aquinas to group exercises by category (there is also an implicit tag for each language allowed) before presenting them to a student, and Tom's `prerequisites` define the exercises that a student should complete before this one. (Prerequisites can, in turn, require further prerequisites, as "nop" does.)

Tom also defines how Aquinas will grade this exercise, stated simply in a `checks` block. This case is straightforward: Aquinas will run `./rop`, as submitted by the student, and Aquinas will check to see if the value it prints to standard out matches the defined value (the secret, which we redacted here). Tom could have also defined expected outputs using regular expressions. The commands involved can be arbitrary, so a series of checks might do much more than run the submitted program.

The optional `services` block defines the network service required for this exercise. This exercise's service is defined by the teacher-written `service-rop.c`, and Aquinas will compile this without any special compiler flags before installing it on its `target` virtual machine. (Services need not be written in the language that is required for solutions; `service-rop.c` will suffice for student work in C, Go, or Python.) The service listens on TCP port 1,032, and so Aquinas will arrange for its firewalls to permit traffic between its `user` virtual machine and its `target` virtual machine on that port. (More on Aquinas's virtual machines later.) Another exercise, canary, requires that its target service is compiled to contain stack canaries; canary thus includes the statement `"compiler_flags": "-fstack-protector"` in its `services` block. Aquinas transparently stores the resulting programs to avoid potentially breaking the exercise by recompiling them later, perhaps with a different compiler.

```
{
  "name":    "rop",
  "summary": "Attack that makes use of ROP",
  "languages": [
    "C",
    "Go",
    "Python"
  ],
  "tags": {
    "Exploitation": true
  },
  "prerequisites": [
    "nop"
  ],
  "checks": [
    {
      "kind": "basic",
      "parameters": {
        "command": "./rop",
        "stdin":    null,
        "stdout":  "[REDACTED]",
        "stderr":   null,
        "exitCode": 0
      }
    }
  ],
  "services": [
    {
      "source": "service-rop.c",
      "port":    1032,
      "publish_binary": true
    }
  ],
  "service_files": [
    "flag"
  ]
}
```

Figure 1: The definition of an Aquinas ROP exercise

All of this serves to make defining exercises easy for Tom. Tom does not need to worry about compiling target service programs; compiling alternate, redacted versions; setting up firewall rules; running graders; or placing files so that they are available to target services. Instead, Tom simply defines his exercises using Aquinas's syntax, and pushes his definitions to Aquinas's Git repository. Aquinas takes care of the rest. Refer to the documentation at https://www.flyn.org/projects/Aquinas for more examples of Aquinas exercise descriptions.

Samantha the student's view

Samantha directly interacts with two facilities provided by Aquinas: its web server and Git repository.

Aquinas's web server presents Samantha with a list of exercises. The order of the list follows each exercise's prerequisites, or Samantha can elect to view the list of exercises as a graph that reflects the prerequisites. This allows Samantha some measure of self-direction, since it illustrates orderings that she would benefit from. Figure 2 depicts part of an exercise graph. Samantha can also review her progress in groups of exercises called courses.

Clicking on any item in the exercise list (or any node in the exercise graph) displays the lesson associated with the exercise. Lessons are either self-contained, or they reference resources such as common textbooks. In either case the lesson includes an exercise assignment along with instructions that remind Samantha how to clone her exercise repository and submit solutions using Git. Samantha selects the exercise named "ROP in C". The top half of Figure 3 shows part of this assignment.

After cloning her exercise repository, Samantha writes a network client that connects to the given host and port before writing a ROP exploit to the resulting socket.

When Samantha submits her solution using Git she will find that Aquinas grades it and provides feedback in the form of any compiler/interpreter errors from her last submission, a pass/fail indication, and, possibly, a hint of how to modify a failed program. Thus Samantha's workflow consists of selecting a lesson, reading the lesson and completing any prescribed practice problems, using Git to clone a repository, writing a solution, using Git to push the solution back to Aquinas, and reviewing the result. The system imposes a generous limit on how many times Samantha can submit her work, primarily to protect disk space. The bottom half of Figure 3 shows what Samantha would see after Aquinas automatically graded her submission, which omitted a `#include` statement.

While Aquinas can host Samantha's code in its own Git repository, it also allows Samantha to submit her solution through third-party Git providers. Aquinas presently supports GitLab, but it could be extended to allow for other options. If Samantha were to select GitLab, she could click a button on an exercise page to fork the Aquinas repository to a private GitLab project. Another button asks Aquinas to grade anything Samantha submitted to that project. Supporting third-party Git providers allows both students and teachers to make use of the features present therein, such as continuous-integration mechanisms. This allows training in industry best practices to accompany education in computer science.

Alex the administrator's view

Aquinas presently divides its facilities across four virtual machines that Alex maintains: the Git repository and grader, the student host, the web server, and the target host.

Git repository and grader. The Git repository and grader runs the standard Git implementation. Each teacher and student has a Unix account on this host, and each user's shell is set to `git-shell`. Aquinas installs hooks that perform work when new material is pushed to a repository:

- **teacher: post-receive** Runs `aquinas-initialize-projects` when a teacher pushes a new exercise or an exercise update. This builds out the infrastructure necessary for the exercise, possibly performing work on all four hosts.
- **student: update** Runs `aquinas-git-update`, which sets a flag that indicates grading is in progress.
- **student: post-receive** Runs `aquinas-git-post-receive`, which executes the grader utility. Once grading is complete, this process cleans up the flag created by the update

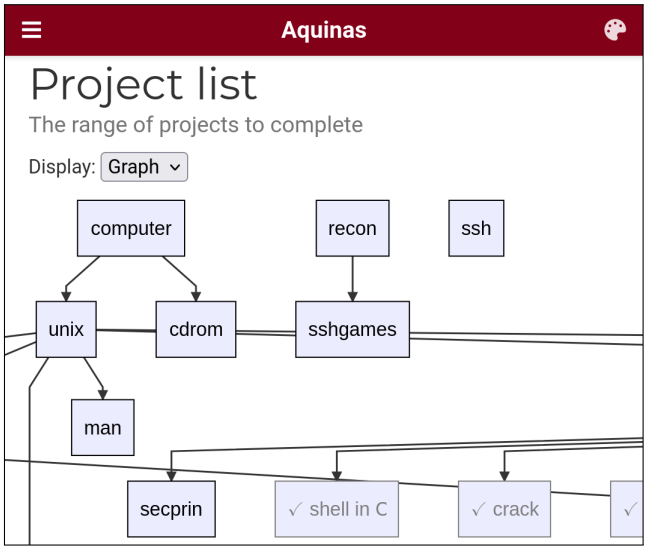

Figure 2: Aquinas displaying the range of exercises in graph form

hook and notifies the web server by calling an HTTP API endpoint.

Thus when Samantha pushes a solution to a Git repository, the latter two hooks trigger a grading job. (In the case of third-party Git providers, clicking on a button on an exercise page triggers a grading job.) Grading jobs proceed by copying code to the student host, possibly compiling the code, and running it according to the exercise's definition (recall the `checks` blocks described above). Running a student submission might cause it to interact with a service on the target host if required by the exercise definition. In any case, the program's outputs flow back to the Git virtual machine, where the grader assesses them. Ultimately, the grader publishes the result to the web server.

Aside from direct use of Git, each request made of the Git virtual machine (including the programs run by Git hooks) passes through a queuing service that serializes requests and ensures each servicing program runs with the correct privileges. For example, the grading Git hook runs as the student who submitted the work, but the queuing service executes the grading process itself with the privileges of a teacher. Aquinas manages these privileges using the standard Unix process and permissions model. The queuing service runs as root, but it applies the principle of least privilege when it executes programs to service requests. The policy that governs who can run what as whom exists as a simple-to-read table in the queuing program's source code.

Student host. The purpose of the student host is to run student submissions in an isolated environment. Students cannot directly log in to the student host, but grading processes running on their behalf can. Though the grading process starts and ends on the Git host, student submissions run on the student host. This involves the grading process, running on the Git host as a teacher, using SSH [29] to connect to the student host as the student user who submitted the work (e.g., Samantha) and providing a set of inputs

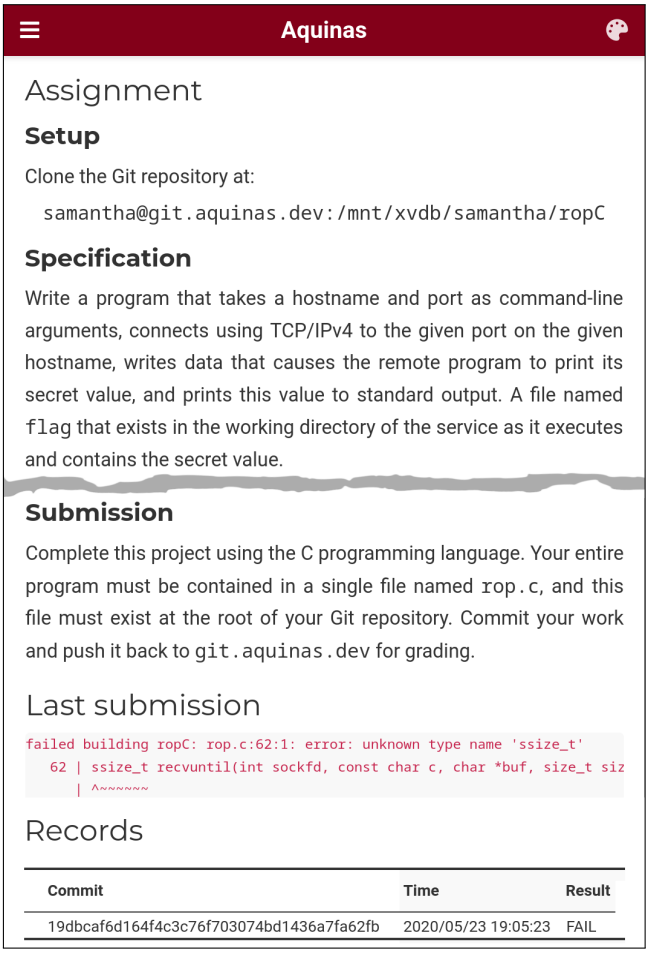

Figure 3: Aquinas displaying a ROP assignment, with portions omitted for brevity; Samantha's submission failed due to a missing `#include`

over the SSH channel to the submitted program. The firewall on the student host prevents nearly all outgoing traffic, with the exception of messages necessary to communicate with the target host (see below). This aims to prevent a student from extracting secrets—such as test inputs—out of Aquinas.

The portion of the grading process that runs on the student host runs with the student's privileges, so the student could subvert it. However, the best Samantha could do is cause this portion of the grading process to report back the correct outputs for the exercise, given the inputs provided by the Git host. This is no different than solving the exercise. The capability to update grades does not exist on the student host, only the capability to send outputs back to the Git repository for evaluation.

Most requests between the Aquinas hosts take place over SSH connections such as with the grading process described above. We chose SSH because of its strong cryptographic protections (including key-based authentication) and its ability to integrate into the Unix model of running programs. Scripts take care of deploying SSH keys within Aquinas, although students like Samantha use a

web-based interface to register the SSH key they use on their development workstations, which is customary. The latency of these connections provides a small cost, but the delays are not noticeable by users in the current Aquinas deployment.

Web server. Aquinas's web server runs a custom HTTP application written in Go. Most requests serviced by the application take the form of an HTTP API. A web browser capable of running WebAssembly is the most common client, but the API allows developers to write other clients or integrate Aquinas's services into other platforms. The components that Aquinas provides as WebAssembly are also written in Go.

The web server can send allowed requests using SSH to the Git repository. For example, the web server might inquire whether an SSH key exists on the Git host for a given student. The use of a customized shell on the Git host restricts what the web server can request. The Git repository processes these requests through its queuing service, just like with its Git hooks.

Target host. The target host is similar to the student host in that it is carefully restricted to include preventing most outgoing messages. The target host runs an Internet service dæmon (inetd) that provides the services required by network-programming exercises, including exploit-type exercises like "ROP in C". Each of these services is carefully confined. As with capture-the-flag exercises, students benefit from the use of inted-style services: such programs interact only with standard input and standard output, so there are a wide range of debugging and analysis approaches available. Students can also transform the programs back into network services using netcat.

API and allowed inputs. Table 1 summarizes the Aquinas HTTP API and other inputs Aquinas conditionally accepts from the Internet. We distribute the full documentation of the HTTP API along with the Aquinas source code.

External requirements. Aside from the four virtual machines, Aquinas relies on a separate mail exchange to send email, and it relies on an external DNS server to resolve the records `www`, `git`, `user`, and `target` to the IP addresses of the correct virtual machines.

5 EVALUATION

Aquinas served as one method used for the continuing education of roughly 100 US Army developers. These developers ranged from soldiers with undergraduate or graduate degrees in computer science to soldiers who until recently had no formal training in the practice of programming but who performed well on an aptitude assessment. More recently, Aquinas has supported undergraduate and graduate courses at University of Wisconsin–La Crosse, including CS120, *Software Design I* (five iterations); CS356, *Software Exploitation*; CS410, *Open Source Development*; CS455, *Fundamentals of Information Security* (two iterations); and CS456, *Secure Software Development* (two iterations).

CS120 serves as an introduction to programming and software design, and it uses the Java programming language. CS356, CS455, and CS456 exercised some of Aquinas's more exotic features, such as constrained software exploitation, the application of static- and dynamic-analysis tools to software, and SELinux policy evaluation.

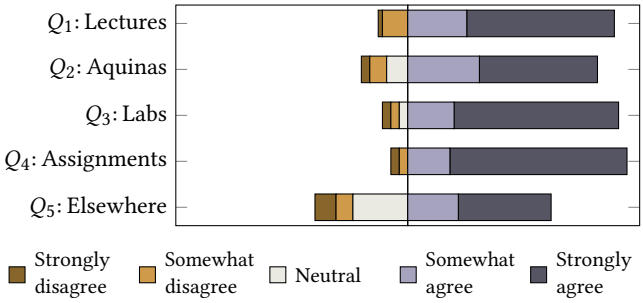

Figure 4: Quantitative survey results

We evaluated Aquinas along two dimensions: student acceptance and resource cost. We measured student acceptance through the use of surveys given to 40 students in CS120 and 22 students in CS356 during the fall semester of 2021. Of the 40 students in the introductory course, 20 were computer science or computer engineering majors, and 20 were not.

5.1 Student Acceptance

Aquinas's impact on learning. We asked our students whether they strongly disagreed, somewhat disagreed, neither agreed nor disagreed, somewhat agreed, or strongly agreed with the following statements:

- Q_1: **Lectures** This course's lectures contributed to your understanding of the material.
- Q_2: **Aquinas** This course's use of Aquinas contributed to your understanding of the material.
- Q_3: **Labs** This course's laboratory exercises contributed to your understanding of the material.
- Q_4: **Assignments** This course's homework assignments contributed to your understanding of the material.
- Q_5: **Elsewhere** I would like to see Aquinas used in other courses that permit automated feedback.

Figure 4 provides the responses to these questions. Students of varying background viewed Aquinas favorably. Since Aquinas delivered the courses' laboratory exercises and homework assignments, the students seemed to indicate that Aquinas contributed to their understanding of the course material at least as much as the lectures. 34 of 56 students (61%) would like to see Aquinas in another course that permits its use. This was more pronounced in the software exploitation course that contained only computer science majors; there 13 of 18 (72%) would like to see Aquinas used elsewhere. Only nine students overall (16%) did not want to see Aquinas in other courses.

Immediate feedback. We asked students "Is the immediate feedback from Aquinas helpful to you when you try to apply a new concept?" Overall, 35 students answered in a positive way, eight answered negatively, and seven were neutral. Many of the students seemed to express that Aquinas did or did not *help them troubleshoot* rather than merely whether the immediate feedback was *helpful*. In the former case, some students were frustrated when Aquinas would provide no error (meaning no compile-time or syntax error) but still rate their submission as failing; better wording from

Table 1: Aquinas HTTP API and other inputs

Mechanism	Authenticated?	Description
HTTP `GET/POST/DELETE`	Yes	Get my data, set my data, delete my account, get my assigned courses, get student rankings, get my grades, and determine if Aquinas is currently grading a submission.
HTTP `GET/POST`	No	Get exercise list, get exercise details, wait for a grading job to complete*, indicate a job completed*, register a new account, and initiate password reset.
SSH *S*`@git/`*S*/*E*	Yes	*S* is a student, and *E* is the name of an exercise. Interact with student exercise using Git/`git-shell`. Triggers grading hooks.
SSH *T*`@git/`*T*`/projects`	Yes	*T* is a teacher. Create or revise exercise using Git/`git-shell`. Triggers exercise-update hook.
SSH *T*`@git/`*T*`/records`	Yes	*T* is a teacher. Interact with student records (grades) using Git/`git-shell`.
SSH `root@any host`	Yes	Allows administrators to interact with Aquinas hosts using a standard shell.

* A UUID replaces the need for client authentication.

Aquinas might help with this. The more advanced students seemed to indicate Aquinas was less helpful with troubleshooting, but that was probably because their exploitation exercises required very precise work. One beginner student stated than an IDE provides better feedback, likely in diagnosing syntax errors; this might be true, but it is orthogonal to the feedback that Aquinas provides.

Use of the Command Line. One of the tradeoffs described in *Courseware as Code* [21] was balancing the benefit of their approach with the tendency of some instructors to favor office-oriented document systems and binary formats over programming tools and markup languages. There the benefit of things like textual revision control and continuous integration outweighed the cost, and the organization was willing to direct that everyone adopt the everything-as-code approach. Aquinas has a reflection of this in its preference of command-line tools over IDEs and in its use of Git. Will students who are new to the practice of programming accept this?

We asked students to "describe how using the command line impacted your learning." Out of 28 qualitative responses from an introductory course, 20 were positive, five were neutral, and three were negative. The more advanced course had stronger results: fifteen of sixteen responses were positive, and one was neutral.

The good, the bad, and the ugly. We asked Q_6: "What aspect of Aquinas did you appreciate?" and Q_7: "What changes to Aquinas might have helped you learn?" Some of the more illustrative responses follow; an asterisk (∗) indicates a response from a CS120 student, and a dagger (†) indicates a response from a more advanced CS356 student.

- Q_6 "The very detailed descriptions for the problems and the fact that it uses a repository to submit work makes me feel more secure then sending code over email and having it potentially not be received by the instructor."†
- Q_6 "Feedback sometimes if you failed. It hosts a bunch of material too so you can try some of them outside of class."†
- Q_6 "I love that it uses the git workflow to upload assignments because it makes it easy to save incremental progress and makes uploading files easier than any other course uses."†
- Q_6 "I liked that there are lessons on each assignment to explain what is going on - each assignment is self-contained in a way, which is nice. I also like the ability to do other assignments on Aquinas and the ability to know pre-reqs for each assignment. The graph/tree visualization is cool as well. Being able to see course grades and standing in the class is also nice."†
- Q_6 "That we had to learn basic git commands."*
- Q_6 "I like the use of the command prompt to learn how to push to Aquinas"*
- Q_6 "The use of git was really cool to see in a 100 level cs class."*
- Q_7 "Improvements to the auto-grading system such as providing error traceback and the input Aquinas ran that caused a program to fail would have been extremely helpful in debugging homework assignments."†
- Q_7 "Having more helpful tips when a 'fail' grade appears"*

Variations on the last response were the most popular answer to Q_7. In some cases, this could be addressed by writing more hints. The compare-type checks described below in §6 should also help. As with Clifton's work [6], we found that a number of beginner students found the use of Git beneficial.

5.2 Resource Cost

Storage space. Aquinas's storage-space cost is small. The system partitions of the web server, Git repository and grader, and target host virtual machines require less than 100 MB each. The `user` virtual machine's system partition is larger due to its programming-language build environments: compilers, interpreters, and supporting libraries costs around 2 GB. The partitions dedicated to student and exercise data on the web server and Git repositories must scale with the number of students and exercises.

Memory and processor use. Aquinas's cost in memory and processor use is light. The `git` and `user` virtual machines require 1,024 and 2,048 MB of memory, respectively, to host large Git repositories and execute development tools and simulation environments. The web server and target host run with 256 MB and 128 MB of memory, respectively. Each virtual machine requires only one CPU core; the most processor-intensive work performed (aside from time-limited infinite loops in student code) is running various compilers.

Sophistication of artifact. Aquinas is presently comprised of 8,657 lines of Go, 1,311 lines of JavaScript, 901 lines of Bourne shell scripts, and 64 lines of C, not including comments or empty lines. This does not include the code in exercises, some of which include the source code of other open-source projects.

```
"checks": [
  {
    "kind": "compare",
    "parameters": {
      "command":    "./guess3",
      "reference":  "guess3C.c,
      "gencmdargs": "generator-cmdargs-guess3.c",
      "genstdin":   "generator-stdin-guess3.c"
    } ...
```

Figure 5: A fragment from the definition of an Aquinas exercise that uses a compare-type check

6 POST-EVALUATION WORK

The automated grading techniques Aquinas supported during the semesters we evaluated required secret test inputs. If such test data were revealed, then students could simply write programs that satisfied the grading script, leaving the set of other inputs unhandled. This meant that a small mistake would leave student work unable to meet the grader's requirement, yet Aquinas could not safely reveal the precise edge case that caused their program to fail. A number of students brought this up in our surveys.

We have implemented an alternative grader that can safely reveal its test inputs, which we call compare-type checks. Instead of defining static inputs and outputs, a teacher can define a reference solution and an input generator, with the latter generating random but well-formed inputs. Compare-type checks present generated inputs both to the reference implementation and the student's solution, and note whether the results are the same for both. A student who submits a failing solution can thus receive the input that caused the failure, as Aquinas will generate another for the next check. Although we have not yet evaluated compare-type checks during a semester, we suspect they will ease student troubleshooting. Not all exercises are well-suited for compare-type checks, but we have already added them to 15 introductory exercises.

Figure 5 provides an exercise definition fragment that defines a compare-type check for a too-high/too-low guessing game. The corresponding file `guess3C.c` contains the teacher's solution; `generator-cmdargs-guess3.c` generates the expected command-line arguments: a single random integer representing a target value; and `generator-stdin-guess3.c` generates the expected standard input: three random integer guesses. Teachers can combine compare-type checks with basic checks. Running randomized compare-type checks repeatedly, followed by basic checks that check boundary conditions and provide clear hints seems to be a good balance.

7 FUTURE WORK

This section lays out the improvements planned for Aquinas. Work during each semester involves addressing student-found issues, and major improvements take place between semesters.

Scalability. Part of making Aquinas scalable means rebuilding Aquinas to exist as a cloud application, as defined by *The US National Institute of Standards and Technology (NIST) Definition of Cloud Computing* [16]. Cloud computing provides for on-demand self service, broad network access, resource pooling, rapid elasticity, and measured service.

An efficient, scalable implementation of Aquinas requires that it support deployment as either virtual machines or containers. Container support will require the reimplementation of some of Aquinas's administrative scripts, which are responsible for grading and other features. The result will be an architecture that can be hosted at a lower cost and in more environments.

We aim to lighten Aquinas' dependence on the local computer of its students and teachers. Infrastructure can become a burden, even when using virtual machines. Technology moves quickly, and problems arise on local machines because of this advance—for example, consider Apple's adoption of their M1 processor, which disrupted the use of the open-source VirtualBox virtualization product on Apple computers. Hosting shell accounts and X2Go-based [19] access within Aquinas would alleviate the problem of configuring student workstations. We also plan to deploy an instance of Aquinas suitable for sharing with others under a software-as-a-service model, alleviating the difficulty of managing private infrastructure.

Vulnerability randomization. Exploiting a vulnerability often involves crafting a careful input, but it is easy to write an exploitive program if the required input is already known. On the other hand, preparing an exploit exercise comes at a high cost due to the precision required. Techniques that would allow Aquinas to generate per-user binaries that minimize the reusability of exploits interest us. Other researchers have proposed per-student exercises; for example, Carlisle pointed out that CyberStakes would sometimes vary problems for each student [3], and Stricklan described techniques for building binary-diverse exercises for a reverse-engineering course [26].

Additional exercises. We have an interest in writing many more exercises for Aquinas. Beyond the use for training and undergraduate and graduate courses, we envision the use of Aquinas as the foundation for outreach to middle- and high-school students. We also hope to add support for more programming languages, such as Rust and JavaScript.

Support for writing exercises. Aquinas already provides a syntax that eases defining exercises, to include necessary infrastructure support. However, writing exercises is presently a task that teachers execute outside of Aquinas, using traditional tools such as text editors. Aquinas takes over when the teacher performs a Git push. Aquinas could support defining exercises from within a browser-based editor.

Robustness and reliability. As a research prototype, Aquinas could be made more robust and reliable in a number of places. We hope to execute a full review of the Aquinas code base, and improve Aquinas's implementation so that it is suitable for use by other researchers and teachers. This work could also provide the basis for course exercises.

Controlled access to progress reports. Aquinas currently operates under an flat authentication model where any teacher can observe the work of any student in the system. Aquinas should follow a more fine-grained authentication model. For example, the student captain of a competitive capture-the-flag team might want to observe the

progress of the team towards solving a set of exercises; Aquinas should permit this without allowing the captain to observe other work. This will be required if a single hosted instance of Aquinas is to support a number of different institutions.

Aquinas has already impacted other open-source projects. Work on Aquinas led to a fix for the Dropbear SSH server concerning exit codes [17], the addition of SELinux as a feature supported by OpenWrt [9], an improvement to the username sizes supported by Dropbear [18], improvements to the LaTeXML converter, and a number of packaging improvements for OpenWrt.

8 CONCLUSION

Aquinas aims to maximally apply an everything-as-code approach to teaching the practice of programming and exploit development using hands-on exercises. Aquinas's use of machine-readable exercise definitions helps teachers define a body of rigorous and robust student exercises. We provided evidence that the consistency provided by Aquinas helps even beginner students make use of the command line and Git, further aiding course management. Aquinas has already served as a tool in five undergraduate and graduate courses, and we have plans to further its use. We plan to continue to develop Aquinas to make it easier to use at other institutions. The source code and documentation for Aquinas is available at https://www.flyn.org/projects/Aquinas, and the canonical instance of Aquinas is at https://www.aquinas.dev/.

ACKNOWLEDGMENTS

Chris Apsey gave a lecture on everything as code in 2017, inspiring early thoughts about Aquinas. Jessie Lass applied Chris's insights to the practice of programming in the CSD-D, and these experiences further solidified early ideas. A number of colleagues provided direct help with the early work on Aquinas, including Will Brattain, Thomas Dignan, Jakob Kaivo, Jessie Lass, and Christian Sharpsten.

REFERENCES

[1] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. ACM, New York, New York, USA, 27–38.

[2] Elie Bursztein, Baptiste Gourdin, Celine Fabry, Jason Bau, Gustav Rydstedt, Hristo Bojinov, Dan Boneh, and John C. Mitchell. 2010. Webseclab security education workbench. In *Proceedings of the 3rd Workshop on Cyber Security Experimentation and Test* (Washington, DC, USA) *(CSET '10)*. USENIX Association, Berkeley, California, USA.

[3] Martin Carlisle, Michael Chiaramonte, and David Caswell. 2015. Using CTFs for an Undergraduate Cyber Education. In *Proceedings of the 2015 USENIX Summit on Gaming, Games, and Gamification in Security Education* (Washington, DC, USA). USENIX Association, Berkeley, California, USA.

[4] Kevin Chung. 2017. Lowering the Barriers to Capture The Flag Administration and Participation. In *Proceedings of the 2017 USENIX Workshop on Advances in Security Education* (Vancouver, British Columbia, Canada). USENIX Association, Berkeley, California, USA.

[5] Kevin Chung and Julian Cohen. 2014. Learning Obstacles in the Capture The Flag Model. In *Proceedings of the 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education* (San Diego, California, USA). USENIX Association, Berkeley, California, USA.

[6] Curtis Clifton, Lisa C. Kaczmarczyk, and Michael Mrozek. 2007. Subverting the Fundamentals Sequence: Using Version Control to Enhance Course Management. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA) *(SIGCSE '07)*. Association for Computing Machinery, New York, New York, USA, 86–90.

[7] Crispin Cowan, Seth Arnold, Steve Beattie, Chris Wright, and John Viega. 2003. Defcon Capture the Flag: defending vulnerable code from intense attack. In *Proceedings DARPA Information Survivability Conference and Exposition* (Washington, DC, USA), Vol. 1. IEEE, New York, New York, USA, 120–129.

[8] Wenliang Du and Ronghua Wang. 2008. SEED: A Suite of Instructional Laboratories for Computer Security Education. *Journal on Educational Resources in Computing* 8, 1, Article 3 (March 2008), 24 pages.

[9] Jake Edge. 2020. OpenWrt and SELinux. *LWN* (Sept. 2020). https://lwn.net/Articles/832876/ [Accessed January 25, 2021].

[10] Robert Fanelli and T.J. O'Connor. 2010. Experiences with practice-focused undergraduate security education. In *Proceedings of the 3rd Workshop on Cyber Security Experimentation and Test* (Washington, DC, USA) *(CSET '10)*. USENIX Association, Berkeley, California, USA.

[11] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, New York, USA, 579–584.

[12] Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology* (Charlotte, North Carolina, USA) *(UIST '15)*. Association for Computing Machinery, New York, New York, USA, 599–608.

[13] Philip J. Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing* (Atlanta, Georgia, USA) *(VL/HCC '15)*. IEEE, New York, New York, USA, 79–87.

[14] Gordon Hunter, David Livingstone, Paul Neve, and Graham Alsop. 2013. Learn Programming++: The Design, Implementation and Deployment of an Intelligent Environment for the Teaching and Learning of Computer Programming. In *Proceedings of the 9th International Conference on Intelligent Environments* (Athens, Greece). IEEE, New York, New York, USA, 129–136.

[15] Kief Morris. 2016. *Infrastructure as Code: Managing Servers in the Cloud* (1st ed.). O'Reilly Media, Inc.

[16] Mell Peter and Grance Timothy. 2011. NIST Special Publication 800–145. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf [Accessed July 9, 2020].

[17] W. Michael Petullo and Matt Johnston. 2018. Strange behaviour [sic] surrounding "ssh -T ..." and non-zero exit. Mailing list thread: https://lists.ucc.gu.uwa.edu.au/pipermail/dropbear/2018q4/002119.html [Accessed January 25, 2021].

[18] W. Michael Petullo and Matt Johnston. 2019. MAX_USERNAME_LEN set too low. Mailing list thread: https://lists.ucc.gu.uwa.edu.au/pipermail/dropbear/2019q1/002146.html [Accessed January 25, 2021].

[19] The X2Go Project. 2022. X2Go. https://www.x2go.org/ [Accessed Jan 2, 2022].

[20] Eric S. Raymond and Tim O'Reilly. 1999. *The Cathedral and the Bazaar* (1st ed.). O'Reilly & Associates, Inc., Boston, Massachusetts, USA.

[21] Julianna M. Rodriguez, Benjamin J. Allison, Christopher W. Apsey, and Todd M. Boudreau. 2020. Courseware as Code: Instituting Agile Courseware Collaboration. *IEEE Security & Privacy* 18, 6 (2020), 59–62.

[22] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. 2016. Build It, Break It, Fix It: Contesting Secure Development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, New York, USA, 690–703.

[23] Inc. Ryzac. 2022. Codecademy. https://www.codecademy.com/ [Accessed Mar 23, 2022].

[24] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. ACM, New York, New York, USA, 552–561.

[25] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. *Implementing SELinux as a Linux Security Module*. Report #01-043. NAI Labs. Revised April 2002.

[26] Christopher Stricklan and TJ OConnor. 2021. Towards Binary Diversified Challenges For A Hands-On Reverse Engineering Course *(ITiCSE '21)*. Association for Computing Machinery, New York, New York, USA, 102–107. https://doi.org/10.1145/3430665.3456358

[27] Erik Trickel, Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna. 2017. Shell We Play a Game? CTF-as-a-service for Security Education. In *Proceedings of the 2017 USENIX Workshop on Advances in Security Education* (Vancouver, British Columbia, Canada). USENIX Association, Berkeley, California, USA.

[28] Joseph Werther, Michael Zhivich, Tim Leek, and Nickolai Zeldovich. 2011. Experiences In Cyber Security Education: The MIT Lincoln Laboratory Capture-the-Flag Exercise. In *Proceedings of the 4th Workshop on Cyber Security Experimentation and Test* (San Francisco, California, USA) *(CSET '11)*. USENIX Association, Berkeley, California, USA.

[29] Tatu Ylonen. 1996. SSH—Secure Login Connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium* (San Jose, California, USA). USENIX Association, Berkeley, California, USA, 37–42.