CS50 Sandbox

Secure Execution of Untrusted Code

David J. Malan
School of Engineering and Applied Sciences
Harvard University
Cambridge, Massachusetts, USA
malan@harvard.edu

ABSTRACT

We introduce CS50 Sandbox, an environment for secure execution of untrusted code. Implemented as an asynchronous HTTP server, CS50 Sandbox offers clients the ability to execute programs (both interactive and non-interactive) written in any compiled or interpreted language in a tightly controlled, resource-constrained environment. CS50 Sandbox's HTTP-based API takes files, command lines, and standard input as inputs and returns standard output and error plus exit codes as outputs.

Atop CS50 Sandbox, we have built CS50 Run, a web-based code editor that enables students to write code in a browser in any language, whether compiled or interpreted, that's executed server-side within a sandboxed environment. And we have built CS50 Check, an autograding framework that supports black- and white-box testing of students' code, leveraging CS50 Sandbox to run series of checks against students' programs, no matter the language of implementation.

We present in this work the pedagogical motivations for each of these tools, along with the underlying designs thereof. Each is available as open source.

Categories and Subject Descriptors

K.3.1 [**COMPUTERS AND EDUCATION**]: Computer Uses in Education—*Collaborative learning, Computer-assisted instruction*; K.3.2 [**COMPUTERS AND EDUCATION**]: Computer and Information Science Education—*Computer science education, Self-assessment*

General Terms

Design, Security, Verification

1. INTRODUCTION

Computer Science 50 (CS50) is Harvard University's "introduction to the intellectual enterprises of computer science and the art of programming" for majors and non-majors alike, a one-semester amalgam of courses generally known as CS1 and CS2. The course is required of majors, but most of the course's students are non-majors. In Fall 2011, enrollment was 607, 76% of whom had no prior CS experience, 18% of whom had taken one prior course, and 6% of whom had taken two or more. The course is taught mostly in C in a command-line environment, with PHP and JavaScript introduced toward term's end in the context of web programming. Weekly problem sets (i.e., programming assignments) demand upwards of 15 hours per week of most students.

In years past, students wrote most of their code on a load-balanced cluster of Linux servers on which they had shells and home directories, primarily using `gcc` and `gdb` to compile and debug. In Fall 2011, students instead used a virtual machine (VM) running on their own Mac or PC. As a result, Internet connectivity was no longer requisite, and students could more easily use graphical editors like `gedit`. Moreover, server load (particularly on nights before deadlines) was no longer an issue, since programs were executed on students' own CPUs. But for the very same reason could technical difficulties no longer be resolved server-side for students by staff. And some netbooks struggled under the weight of a VM.

For Fall 2012, then, we wanted to craft an alternative, not to replace the VM but to supplement it. Our objective was to provide students with a lighter-weight, web-based mechanism for trying out code, particularly in lectures and sections (i.e., recitations), where it's much faster to open a browser than boot a VM. The VM, meanwhile, would still be used by students for larger-scale projects that warrant a full-fledged command-line environment or GUI.

We did not wish to sacrifice C, though, for the sake of browser-based coding. Whereas languages like JavaScript and Python can be interpreted client-side using JavaScript itself [13, 16], compilers for languages like C are not so easily ported. Server-side execution of code would therefore be necessary.

Meanwhile, we faced an orthogonal problem. Over the years, the course has written a variety of scripts (e.g., in Perl and Python) with which the course's teaching fellows (TFs) could test the correctness of students' submissions. But those scripts were generally quite fragile, with unanticipated corner cases sometimes causing all tests to fail. Moreover, because the scripts were custom-written for specific assignments without any unifying framework, both writing and running them was rather laborious. The TFs, too, would generally run students' code under the TFs' own UIDs, and so there was always a risk of buggy or malicious code wreaking havoc in TFs' accounts. We've long wanted some form

SIGCSE'13, March 6–9, 2013, Denver, Colorado, USA.

of sandboxing for students' code, then, simple enough that TFs aren't disinclined to use it.

We thus set out to tackle both problems at once. In the sections that follow, we present our solution. CS50 Sandbox is our environment for secure execution of untrusted code. In Section 2, we present the design goals therefor. In Section 3, we present CS50 Sandbox's implementation details, including its front end (an HTTP-based API) and back end (a Linux-based cluster). In Section 4, meanwhile, we introduce CS50 Run, a web-based code editor built atop CS50 Sandbox that enables students to write code in a browser that's compiled and executed server-side. In Section 5, we introduce CS50 Check, an autograding framework built into CS50 Sandbox that enables students and instructors alike to assess the correctness of code. In Section 6, we present future work and conclude.

2. DESIGN GOALS

Even though CS50 happens to introduce students to C, along with PHP and JavaScript, we wanted our sandboxed environment to be language-agnostic so that courses besides ours could leverage the platform. Moreover, we needed it to withstand execution of buggy (if not adversarial) code, lest infinite loops or fork bombs throttle the system. And we wanted the environment to accommodate the course's style of homework, fairly complex programs that can often be designed in any number of ways, rather than have to alter the course's workload to accommodate technology. And so we set out with these design goals in mind:

- support interactive and non-interactive execution of programs written in any language, particularly those commonly used in introductory courses (e.g., C, C++, Java, JavaScript, OCaml, Perl, PHP, Python, Ruby, Scheme, et al.);
- support build tools like `ant` and `make`;
- support standard error, input, and output;
- limit consumption of CPU cycles, disk space, file descriptors, and RAM;
- restrict network access, both inbound and outbound;
- prevent reading and writing of unsandboxed files; and
- prevent denial-of-service attacks.

Before proceeding to implement, we vetted existing solutions. In the web-based space are ideone [14], codepad [15], CodeEval [2], and compilr [3], all of which support browser-based compilation and interpretation of code. But only the last supports interactive execution of programs, though it lacks an API. Even so, we would want more control over our environment than a third party would likely allow. In particular, we would want it to behave identically to our VM, with the same OS, binaries, and libraries installed on both, so that students' programs behave the same, no matter the context.

In the autograding space, related works abound [7–9], although many are now dated and don't support many languages. Most closely related is Web-CAT [1], perhaps the best known, but Web-CAT doesn't sandbox code to the degree that we sought. Nor does it support interactivity or clustering. Unit-testing frameworks also abound for C (and other languages), but none offer the simplicity that we wanted for the course's earliest projects especially. Autolab [5], meanwhile, is the closest incarnation of our own vision for sandboxing but isn't yet available as open source. It also spawns entire virtual machines for sandboxing tests, overhead that we wished to avoid.

3. IMPLEMENTATION DETAILS

At a high level, CS50 Sandbox is a web-based black box that allows for secure execution of untrusted code. Its inputs include files (e.g., `hello.c`), commands (e.g., `./a.out`), signals (e.g., `SIGINT`), and standard input. And its outputs include exit codes plus standard output and error.

CS50 Sandbox is not meant to be used by humans directly, as via a GUI. Rather, it accepts inputs via HTTP through API calls, and it returns outputs via HTTP as JSON objects [4]. Clients (e.g., CS50 Run, per Section 4) can issue those calls via Ajax (if sandboxing non-interactive programs) or via WebSocket [17] (if sandboxing interactive programs).

CS50 Sandbox is currently packaged for Red Hat-based systems (among them CentOS, Fedora, RHEL, and Scientific Linux) but could be ported to Debian-based systems as well. It is available as open source at `http://cs.harvard.edu/malan/`.

3.1 Front End

CS50 Sandbox's front end is an HTTP-based API, among whose endpoints are `/upload` and `/run`.

3.1.1 /upload

To upload one or more files to the environment, a client need only POST them to `/upload` (as `multipart/form-data`). CS50 Sandbox's response will be a JSON object with two keys: `files`, whose value is an array of the filenames POSTed, and `id`, whose value is a unique identifier for the collection of files, per the below.

```
{
 "files": [String, ...]
 "id": String
}
```

3.1.2 /run

To execute a non-interactive command (e.g., `clang hello.c`) inside of CS50 Sandbox, a client need only POST to `/run` a JSON object with two keys: `cmd`, whose value is the command line to execute, and `sandbox`, whose value is a JSON object with one key, `homedir`, whose value is the unique identifier for a previously uploaded collection of files, per the below.

```
{
 "cmd": String,
 "sandbox": { homedir: String }
}
```

CS50 Sandbox will then copy those files into a temporary `$HOME`, in which `cmd` will be executed. Upon successful execution, the server will respond with a JSON object of the form

```
{
 "code": Number,
 "sandbox": String,
 "script": String,
 "stderr": String,
 "stdout": String
}
```

where `code` is the command's exit code, `sandbox` is a unique identifier for the temporary `$HOME` so that the client may execute subsequent commands (e.g., `./a.out`) in the same sandboxed environment, `script` is the command's terminal output (with standard error and standard output interwoven), `stderr` is the command's standard error, and `stdout` is the command's standard output

To execute an interactive command that expects standard input (e.g., `./a.out`) inside of CS50 Sandbox, a client can instead establish (or emulate, as with socket.io [12]) a WebSocket with CS50 Sandbox and emit a `run` event, whose payload is also a JSON object whose keys are, as before, `cmd` and `sandbox`. CS50 Sandbox will then execute `cmd`, emitting `stdout` and `stderr` events over the socket as standard output and error, respectively, are generated. CS50 Sandbox will also emit a `stdin` event when it detects that `cmd` is blocking for input so that the client knows to provide. Once `cmd` exits, CS50 Sandbox will emit an `exit` event, whose payload is an exit code plus standard output and error, along with a unique identifier for the `$HOME` used. A client can also interrupt `cmd` prematurely by emitting a `SIGINT` event.

3.2 Back End

Underneath the hood, CS50 Sandbox is an HTTP server written in JavaScript for Node.js [11], an open-source runtime built atop V8 [6] whose event-driven architecture is well-suited for handling asynchronous events (e.g., interactive standard I/O). Moreover, Node.js supports non-blocking I/O, which means that execution of one (slow-running) `cmd` won't block that of another. The architecture supports clustering, whereby CS50 Sandbox can be run on any number of load-balanced servers.

Upon receiving an HTTP request whose endpoint is `/upload`, CS50 Sandbox saves each file POSTed asynchronously to disk in a directory whose name serves as the unique identifier for that collection of files, which is then returned to the client.

Upon receiving an HTTP request whose endpoint is `/run` or a `run` event over a socket, CS50 Sandbox sets up a "sandbox," a temporary `$HOME` in which a `cmd` will be executed. It copies a collection of files (identified by an `id`) into that sandbox, creates a new user (whose username is also `id` and whose password is locked), and then assigns `$HOME` to that user. It then executes `cmd` on behalf of that user, via `sudo`. But it first wraps `cmd` with a call to `nice`, so as to prioritize its execution below the HTTP server itself. It also wraps `cmd` with a call to `seunshare`, which further confines `cmd` to the temporary `$HOME` (and its own `/tmp`) within an alternate "context," a feature of SELinux [10] that restricts processes' behavior, blocking access to `/etc/passwd`, `/proc`, and more. Finally, `cmd` is wrapped further with a call to `strace`, which allows CS50 Sandbox to detect writes on standard output and error (as well as reads on standard input).

So that `cmd` can only consume its fair share of resources, CS50 Sandbox imposes constraints using `pam_limits`, which caps processes' consumption of CPU cycles, disk space, file descriptors, RAM, and more on a per-user basis, ergo the creation of a user per sandbox. Meanwhile, `iptables` restricts network access. CS50 Sandbox's own HTTP server additionally monitors wall time, killing long-running processes (in addition to fork bombs).

Once `cmd` exits, CS50 Sandbox responds to the client and tears down the sandbox after waiting a few minutes for any subsequent `cmd`.

We happen to run CS50 Sandbox on Amazon EC2, using Amazon ELB for load-balancing and Amazon S3 for shared storage, but the environment can just as easily be run locally on one or more servers, using DNS or layer-3 alternatives for load-balancing and NFS or SMB for shared storage as needed. Indeed, the environment can be installed with a single command (`yum`).

4. CS50 Run

CS50 Run is a web-based code editor that lives at `https://run.cs50.net/`. Built atop CS50 Sandbox, CS50 Run enables students to write, within a browser, code in any language, execution of which happens server-side. Implemented in JavaScript, HTML, and CSS, CS50 Run leverages CS50 Sandbox's API to create an illusion that execution is local, even providing students with a terminal window in which they can not only see standard output and error but provide standard input as well. Via a drop-down can students select the language in which they wish to write code, each of which maps to a particular `cmd` (e.g., `clang` for C). Each `cmd` is executed automatically for students, based on that drop-down's selection, but each `cmd` is displayed in the illusory terminal window so that students can see exactly what's happening under the hood. Lines of code, meanwhile, are automatically numbered and syntax-highlighted. And via a button can students save revisions of their code, so that they can roll back in time and recover from crashes. Figure 1 depicts.

Not only does CS50 Run enable students to work on small programs in lectures and sections without booting a VM, it also empowers students to check their code's correctness instantly via integration with CS50 Check.

5. CS50 Check

CS50 Check is an autograding framework built into CS50 Sandbox that enables black- and white-box testing of students' code. Like CS50 Sandbox itself, its front end is an HTTP-based API. Its back end, meanwhile, comprises series of checks (i.e., tests) that are written in JavaScript (but can check the correctness of programs written in any compiled or interpreted language).

5.1 Front End

To run some program (i.e., `cmd`) through a series of checks, a client (e.g., CS50 Run, which itself uses CS50 Check to provide students with browser-based feedback) need only POST the program's file(s) to CS50 Sandbox via `/upload`, at which point it can then contact `/check`, CS50 Check's sole endpoint, to initiate testing.

5.1.1 /check

To run a series of (instructor-defined) checks against a previously uploaded collection files, a client need only POST to `/check` a JSON object of the following form, where `checks`

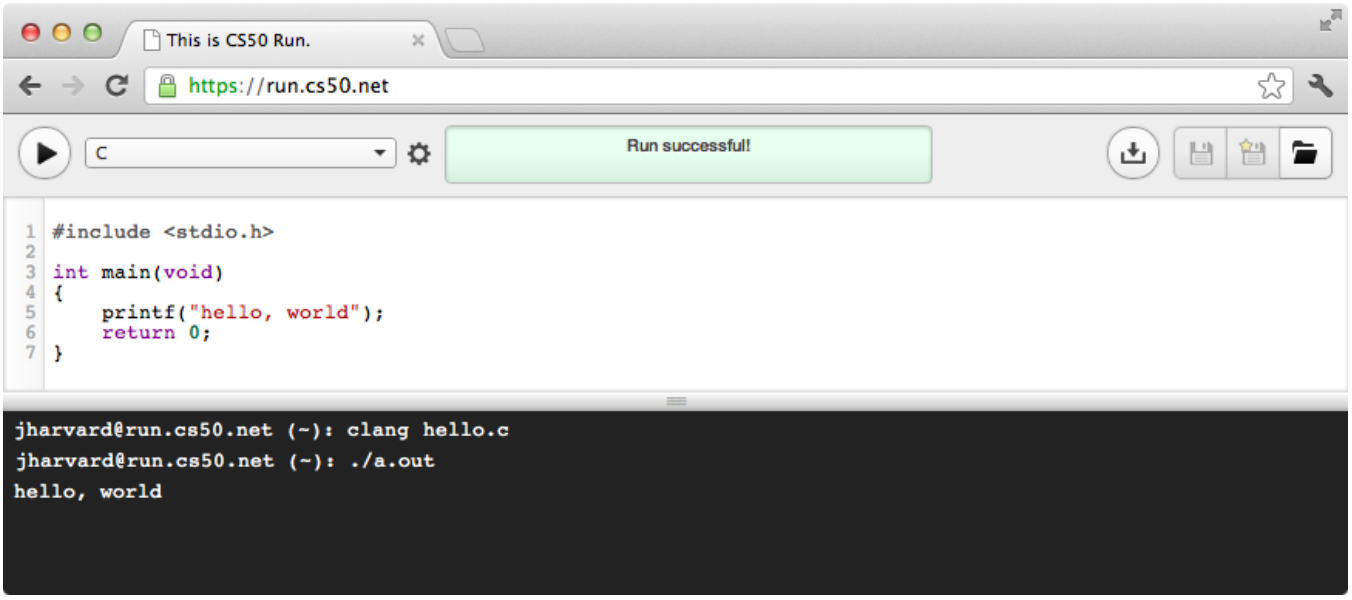

Figure 1: This is CS50 Run, a web-based code editor built atop CS50 Sandbox that enables students to write programs in any language within a browser.

is a unique identifier for the series of checks, and `sandbox` defines (as it also does for `/run`) a sandbox to use for the checks' execution, where `homedir` is the unique identifier for a previously uploaded directory of files, a copy of which will be mounted as `$HOME` for each of the checks in the series.

```
{
 "checks": String,
 "sandbox": { homedir: String }
}
```

Upon execution of checks, the server will return a JSON object of the form

```
{
 "results": Object
}
```

where `results` is an object, each of whose keys is the unique identifier for a check in the series, the value of which is an object of the form

```
{
 "dependencies": [String, ...],
 "description": String,
 "result": Boolean,
 "script": [Object, ...]
}
```

where `dependencies` is an array, each of whose elements is the unique identifier for another check on which the check depends, `description` is a human-friendly description of the check, `result` is either

- `true`, which signifies that the check passed,
- `null`, which signifies that the check was not executed because of a failed or erred dependency, or
- `false`, which signifies that the check failed,

and `script` is an array of objects, each of the form

```
{
 "actual": { type: String, value: String },
 "expected": { type: String, value: String }
}
```

where `expected` describes what was expected, and `actual` describes what actually occurred, if anything, up until the point when the check passed or failed. For instance, this response indicates that two checks (`compiles` and `runs`) passed:

```
{
 "results": {
  "compiles": {
   "dependencies": [],
   "description": "Does hello.c compile?",
   "result": true,
   "script": [Object, ...]
  },
  "runs": {
   "dependencies": [
    "compiles"
   ],
   "description": "Does a.out run?",
   "result": true,
   "script": [Object, ...]
  }
 }
}
```

By contrast, this response indicates that one check passed whereas another check failed (because of an unexpected exit code):

```
{
 "results": {
  "compiles": {
   "dependencies": [],
   "description": "Does hello.c compile?",
   "result": true,
   "script": [Object, ...]
  },
  "runs": {
   "dependencies": [
    "compiles"
   ],
   "description": "Does a.out exit with 0?",
   "result": false,
   "script": [
    ...,
    {
     "actual": {
      "type": "exit",
      "value": 1
     },
     "expected": {
      "type": "exit",
      "value": 0
     }
    }
   ]
  }
 }
}
```


Similarly does this response indicate that one check passed whereas another check failed (because of unexpected standard output):


```
{
 "results": {
  "compiles": {
   "dependencies": [],
   "description": "Does hello.c compile?",
   "result": true
  },
  "runs": {
   "dependencies": [
    "compiles"
   ],
   "description": "Does a.out print \"hello, world\"?",
   "result": false,
   "script": [
    ...,
    {
     "actual": {
      "type": "stdout",
      "value": "goodbye, world"
     },
     "expected": {
      "type": "stdout",
      "value": "hello, world"
     }
    }
   ]
  }
 }
}
```


And this response indicates that two checks failed, one of which wasn't even executed because of its dependency on the other:


```
{
 "results": {
  "compiles": {
   "dependencies": [],
   "description": "Does hello.c compile?",
   "result": false,
   "script": [
    ...,
    {
     "actual": {
      "type": "exit",
      "value": 1
     },
     "expected": {
      "type": "exit",
      "value": 0
     }
    }
   ]
  },
  "runs": {
   "dependencies": [
    "compiles"
   ],
   "description": "Does a.out run?",
   "result": null,
   "script": []
  }
 }
}
```


5.1.2 Back End

On the back end, series of checks are implemented in JavaScript as Node.js "modules," objects whose keys are checks' names, whose values are anonymous functions that return functional tests. Checks themselves are implemented as "chains," sequences of asynchronous methods, each of whose execution is delayed until its preceding sibling invokes a callback. These chains hide from instructors the underlying implementation details of asynchronous code, which is not always easy to write. After all, the objective of CS50 Check is to challenge students to pass checks, not instructors to write them! Below is a representative series of two checks, the first of which checks, by running `clang` (with `.run`, which runs a `cmd` in a sandbox), whether `hello.c` compiles with an exit code of 0, the second of which checks whether the resulting `a.out` says hello, as per the `RegExp` (which could alternatively be a `String` specifying a file on disk against which to compare standard output).

```
module.exports = {
 "compiles": function(check) {
  return check("Does hello.c compile?")
  .run("clang hello.c")
  .exit(0);
 },
 "runs": ["compiles", function(check) {
  return check("Does a.out print \"hello, world\"?")
  .run("./a.out")
  .stdout(/^hello, world$/)
  .exit(0);
 }
};
```

CS50 Check includes support for other "links" in checks' chains as well, including `.diff` (which compares two files for differences), `.exists` (which checks whether a file exists), `.stderr` (which checks students' standard error against a `RegExp` or file), and `stdin` (which checks whether a program is blocking for standard input).

CS50 Check also supports specification of dependencies. Indeed, in the above, the value of `runs` is actually an array, the first of whose values is `"compiles"`, which prescribes that the second check should be executed only if the first of the two passes.

6. FUTURE WORK, CONCLUSION

CS50 Sandbox is an environment for secure execution of untrusted code that we designed underneath both CS50 Run, a web-based code editor that enables students to write code in any compiled or interpreted language, and CS50 Check, an autograding framework that enables students and instructors alike to assess the correctness of code.

In Fall 2012, we will deploy CS50 Sandbox not only to our own undergraduates but to as many as 120,000 edX students as well. Coupled with CS50 Run and CS50 Check, CS50 Sandbox will ultimately enable students to work on problems within browsers during lectures and sections, without the need for client-side VMs. Moreover, it will enable us to embed alongside curricular materials real-time coding exercises on which students will receive instant feedback.

So that the infrastructure scales dynamically with load, we may eventually utilize Amazon Auto Scaling, which will allow us to define thresholds for load, beyond which additional servers will be spawned automatically. We anticipate adversarial attacks on the system, and so we plan to harden the system over time as we detect weaknesses. In particular, we expect to write our own SELinux policy with which to confine each `cmd` all the more. We plan, too, to augment CS50 Check's pool of methods to support measurement of resources consumed during runtime (e.g., CPU cycles and RAM).

Meanwhile, we expect that the thousands of submissions that CS50 Sandbox will collect over time will yield insights into the process by which students write code and respond to continual feedback.

7. ACKNOWLEDGMENTS

Many thanks to Dan Walsh of RedHat and to Glenn Holloway, Joseph Ong, Mike Tucker, Nate Hardison, Rob Bowden, and Tommy MacWilliam of Harvard for their assistance with this work.

8. REFERENCES

[1] Anuj R. Shah. Web-CAT: A Web-based Center for Automated Testing. Master's thesis, Virginia Polytechnic Institute and State University, 2003.

[2] CodeEval Inc. CodeEval. http://www.codeeval.com/.

[3] Compilr Inc. compilr. https://compilr.com/.

[4] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt?number=4627.

[5] Dave O'Hallaron et al. Autolab. http://autolab.cs.cmu.edu/.

[6] Google Inc. V8 JavaScript Engine. http://code.google.com/p/v8/.

[7] J. A. Harris, E. S. Adams, and N. L. Harris. Making program grading easier: but not totally automatic. *J. Comput. Sci. Coll.*, 20(1):248–261, Oct. 2004.

[8] D. Jackson. A semi-automated approach to online assessment. In *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, ITiCSE '00, pages 164–167, New York, NY, USA, 2000. ACM.

[9] D. Jackson and M. Usher. Grading student programs using assyst. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, SIGCSE '97, pages 335–339, New York, NY, USA, 1997. ACM.

[10] James Morris. SELinux Project Wiki. http://selinuxproject.org/.

[11] Joyent, Inc. Node.js. http://nodejs.org/.

[12] LearnBoost, Inc. Socket.IO. http://socket.io/.

[13] Scott Graham. Skulpt. http://www.skulpt.org/.

[14] Sphere Research Labs. ideone. http://ideone.com/.

[15] Steven Hazel. codepad. http://codepad.org/.

[16] Syntensity. Python on the Web. http://syntensity.com/static/python.html.

[17] World Wide Web Consortium. The WebSocket API. http://dev.w3.org/html5/websockets/.