NERDS: A Non-invasive Environment for Remote Developer Studies

Joseph Lewis
jlewis23@umd.edu
University of Maryland
College Park, Maryland, USA

Kelsey R. Fulton
kelsey.fulton@mines.edu
Colorado School of Mines
Golden, Colorado, USA

ABSTRACT

Given the difficulties of secure development, studying software developers remains pivotal. However, conducting these studies remains a pain point for the security community as recruitment and retention can be incredibly difficult. In this work, we aim to make conducting security studies with software developers easier by building **NERDS**: a **N**on-invasive **E**nvironment for **R**emote **D**eveloper **S**tudies. **NERDS** allows for conducting remote studies while still providing researchers with a controlled environment. We describe our experiences building and deploying **NERDS** in two distinct secure software development studies. Our lessons learned can provide valuable insight to other researchers wanting to utilize **NERDS**. We provide **NERDS** as an open-source system.

CCS CONCEPTS

• **Security and privacy** → **Usability in security and privacy**; **Social aspects of security and privacy**.

KEYWORDS

Study methodology, Secure software development, Usable security and privacy

1 INTRODUCTION

Secure software development remains a difficult task as evidence by the many vulnerabilities introduced on a regular basis [6, 14, 18]. Developers are a key cause of these vulnerabilities [2, 11, 16, 17, 20, 25]. As such, in order to address these vulnerabilities, we need to understand how and why these vulnerabilities occur and how developers think about secure development.

However, conducting these studies remains a challenge. Ecological validity is difficult to achieve as professional development environments are hard to simulate [2, 19]. Recruiting and retaining this specialized population, software developers, can be difficult because they can be hard to find and contact, and they often participate in these studies outside of work hours for compensation

lower than their normal rates [16, 21]. Additionally, the sometimes frustrating nature of code writing can lead to high dropout rates and small sample sizes for these studies [1].

Given this, it is important to build studies and their infrastructure to support recruiting and retaining as many participants as possible. To this end, we built **NERDS**: a **N**on-invasive **E**nvironment for **R**emote **D**eveloper **S**tudies. **NERDS** expands on a prior infrastructure [22] by focusing on the ability to scale studies and easily adapt to different study designs.

NERDS is built using Docker containers to maximize adaptability, scalability, and resource efficiency. Researchers can customize **NERDS** using different Docker containers for different studies. Participants interact with **NERDS** entirely through their browser, meaning they do not need to install or download anything to participate. Researchers are able to collect ample data such as participant interactions, internet searches, and time focused on the study. We have deployed **NERDS** in two studies: one completed with 141 participants using the Python programming language and one in-progress with 25 participants using the C language.

In this paper, we detail our experience with designing, building, and deploying the **NERDS** system. We also provide lessons learned from deploying **NERDS**, in an effort to aid researchers who may want to utilize **NERDS** in the future. The code for **NERDS** is open-source, and details and documentation for using the system can be found at our website https://joelewiss.github.io/nerds/.

2 RELATED WORK

Prior research has explored making developer studies better through recruitment, measuring participant skill, and study deployment.

Recruitment. Prior work has explore improving software development studies through understanding the validity of common recruitment approaches. Researchers have studied common recruitment approaches such as freelance marketplaces, GitHub users, and social media to determine their validity in recruiting developers [3, 4, 26]. To understand how students and freelancers compared to professionals, Naiakshina et al. compared the two populations, uncovering that developers performed better overall, but that treatment effects held the same for all populations [15]. Recently, researchers compared and contrasted a number of common recruitment venues finding that CS students are viable alternatives for developers [13, 23]. Finally, Serafini et al. interviewed 30 developers to understand why they participate in studies, finding that the length, compensation, and topic of the study played an important role [21].

Measuring developers' skill. To improve the quality of participants, prior work has developed scales and survey questions to

evaluate the programming and security skills of potential participants. To measure programming experience, researchers built multiple scales utilizing self-efficacy and programming exercises to measure skill [5, 10]. To aid in recruitment, Danilova et al. built a screening questionnaire [8] and explored time limits as a way to improve efficiency [7]. Focusing on security, Votipka et al. built a scale to measure participants' security self-efficacy [24].

Study design. Most related to our work is prior research exploring alternatives to in-person lab studies. Stransky et al. built the Developer Observatory to conduct remote studies using Python [22]. The Developer Observatory system allows participants to use their web browser to complete a series of programming tasks. They found that remote studies conducted using the Developer Observatory were easier and faster than in-person lab studies, but were slightly limited in data collection. **NERDS** utilizes the Developer Observatory as a starting point for our work, but we expand the system by improving scalability and adaptability.

More recently, Huaman et al. built OLab [12], another platform for conducting remote developer studies. Rather than providing a Python programming interface, OLab provides an entire desktop environment that can be configured to support any programming environment. While this is an expansive system, it requires multiple servers to host (due to the use of Kubernetes) and is a much larger (and heavier) system than is needed for most code-focused studies. Additionally, the OLab system is not open-sourced and has only been tested with loads of less than 25 participants. While we drew some inspiration from the design of OLab, **NERDS** is lighter and allows for conducting studies with hundreds of participants at a time.

3 NERDS: ITERATION 1

To develop **NERDS**, we modified the existing Developer Observatory infrastructure with the hope of improving the scalability and adaptability of the system [22]. In this section, we describe the system requirements, system design, and the implementation and deployment results of this first system. We highlight where **NERDS** differs from the Developer Observatory.

3.1 Requirements

We describe the requirements of **NERDS** across four axes: participant experience, experimental design, technical, and study-specific. We specifically highlight changes made to the system in order to address the problems in the original system [22].

3.1.1 Participant experience requirements. The first set of requirements focus on ensuring that participants have the best experience possible, as participant retention is a key challenge of conducting developer studies [1]. Given this, we want to make participating in the study as easy as possible for participants. To this end, the system should be intuitive, designed with developers in mind, and should not require participants to install any additional software on their device (***non-invasive***). Rather, everything should be self-contained and accessible through a single, provided link, only requiring potential participants to have an up-to-date browser installed. Further, given the length and frustration of these studies, participants should also be able to leave and return to the study at arbitrary times and

pick up from where they left off (***flexibility***). This gives the participant flexibility to complete the study on their own time, something not possible during an in-person lab study. Additionally, participants should be able to skip tasks they are struggling with and revisit any previous tasks (***skip-revisit***).

3.1.2 Experimental design requirements. The second set of requirements focus on maximizing the validity of the research. To ensure internal validity, the system should allow for randomized assignment to conditions and randomized presentation of tasks (***randomization***). Additionally, the system should provide data collection and environmental control to a level as close as possible to an in-person lab study (***control***). This means the system should provide the same programming tools, development environment, and programming experience for all participants. To ensure external validity, the system should mirror a real-world programming environment as closely as possible, which means allowing participants to utilize their own devices and emulating the input and output of development environments as closely as possible (***real-world***). Finally, in order to garner the best results possible, the system should record as much data as possible so it can mimic the advantages of an in-person lab study, while not being over intrusive in its data collection (***data collection***). The system needs to record data about the participants programming progress throughout the tasks. This includes, but is not limited to, the participant's responses to the programming tasks, any associated compilation output or errors, and each iteration of code the participant tests. This allows a wide range of studies to be conducted on the system. In order to promote ethical research and limit privacy concerns, the collected data should be recorded securely with an associated pseudonymous identifier (***ethics***).

3.1.3 Technical requirements. The third set of requirements focus on the technical needs to facilitate studies through a system like this. First, the system should be as secure as possible in multiple respects. Participants should be fully isolated from each other and the system itself (***isolation***). There should be no way a participant can view or change the responses of another participant. Participants should not be able to significantly degrade the experience for another participant through an attack like a denial of service (DoS). Second, the system should be scalable, meaning many participants should be able to use the system concurrently without any noticeable performance degradation (***scalability***). More concurrent participants means more opportunities to collect data. Scalability was a notable limitation of the original Developer Observatory. [22], so we aim to address this in our design. Finally, the system should be easy to adapt to new study designs (i.e. different programming languages, different types of coding tasks, adding new measurements or data collection)(***adaptability***). This promotes future use and adoption of the system, maximizing it's utility for researchers. While this is not an entirely distinct goal from the Developer Observatory [22], as they aimed to make their system adaptable to all Python studies, our system takes this requirement a step further by making **NERDS** adaptable to *any* type of code-focused study. We demonstrate the ***adaptability*** of **NERDS** through the deployment of two very different studies which we discuss in Sections 3.2 and 4.2.

3.1.4 Study-specific Requirements. Finally, we describe the study-specific requirements of this system. The first study we utilized **NERDS** for (Study 1) required participants to complete two tasks in the Python language. This study had six total conditions. Three of the conditions were focused on study methodology: writing code from scratch (*Write*), reading existing code and finding and explaining vulnerabilities and bugs (*Read*), or finding and fixing vulnerabilities and bugs in existing code (*Fix*). Then, within each of these methodologies, participants were assigned to use one of two different libraries. Only participants in the *Write* and *Fix* conditions should be able to edit and run code. Additionally, participants in the the *Write* condition should be presented with function headers but not completed code, where participants in the *Read* and *Fix* conditions should be presented with completed code. Finally, participants needed to be redirected to a survey after completing the tasks.

3.2 Design

To meet these requirements, as mentioned in Section 2, we modified the Developer Observatory [22]. Our modifications focused on improving the ***scalability*** and ***adaptability*** while still meeting the other requirements, as discussed above. We will give an overview of our system design and how this satisfies our above requirements, highlighting key differences between the original Developer Observatory and our modified version. Descriptions of how each requirement is satisfied by our design can be found in Table 1.

The system has two main components: the study infrastructure and the participant instances. The main function of the system is to provide every participant in a study with their own *instance* containing a web-based programming environment. For this study the instance was a Python programming environment mirroring the original deployment [1].

3.2.1 Study Infrastructure. The study infrastructure is responsible for managing participant instances and data storage. In the previous design of the Developer Observatory, the services and the participant instances were run on virtual machines hosted on Amazon's EC2 platform. Instead, **NERDS** uses Docker[1] to host services and participant instances. Docker is a open source software for creating and running containers. Containers are less resource intensive while providing some of the same benefits of virtual machines, such as ***isolation***. Each infrastructure component from the previous system is now its own container image hosted with Docker. Containerisation gives multiple benefits to our system, including quicker startup times and less overall resource usage, increasing the ***scalability*** of **NERDS**. There are a total of 7 services needed for the system's backend, plus one helper script that runs on the host where participant instances are hosted. Below is a description of each service and its purpose in the system.

- **Router (`nginx`)** Routes all incoming connections to either the study infrastructure or the participant instances. Runs Nginx, a popular reverse proxy software. In the previous Developer Observatory system, participants were redirected to an AWS EC2 instance on a separate domain that lacked HTTPS.
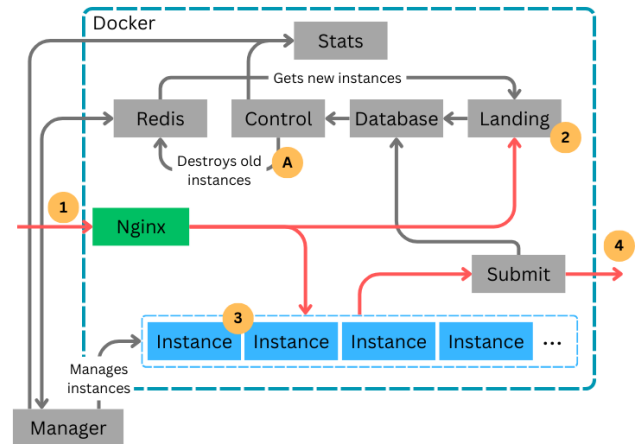
Figure 1: Diagram of the NERDS system. Grey lines are data paths throughout the system. Red lines is the path a participant takes through the system. Yellow labels match to the participant flow described in Section 3.2.1

Instead, all participants are proxied to their instances on the same domain as the study infrastructure, maintaining HTTPS security throughout the study.

- **Manger** Python script on the host that manages starting and stopping containers. In the previous Developer Observatory system, this communicated with Amazon's EC2 API. Instead, containers are started through Docker, making the process faster. Maintains a pool of containers ready for new participants (ready pool). Pool size can be configured to meet expected demands.
- **Landing (`landing`)** PHP application for handling all new participants. Validates user information and system conditions and gets a new instance from the `manager`. Presents the user with intro information on how to complete the study.
- **Submit (`submit`)** Handles communication between participant instances and infrastructure. Stores participant data in db service. Redirects participants to an exit survey on completion.
- **Database (`db`)** PostgreSQL database. Used to store all information relating to the study including conditions, running participant instances, and participant data.
- **Redis (`redis`)** Maintains a queue of started instances ready to be given to users. Main communication path between `landing` container and `manager` script.
- **Control (`control`)** Checks for abandoned instances and signals `manger` to destroy them through `redis`. Reports statistics to `stats` service.
- **Stats (`stats`).** Aggregates statistics from services and displays graphs for study administrators.

Participants completing a study on **NERDS** would go through the following steps inside of the study infrastructure:

(1) A participant completes a pre-survey (consent or screening), separate from the system. From this survey, the participant is

	Requirement	Description
Participant experience	*non-invasive*	Participate through a browser. No installs or downloads necessary.
	flexibility	Uses cookies to allow rejoining within allotted time period.
	skip-revisit	Provides skip/previous buttons.
Experimental design	*randomization*	Arbitrary conditions configurable by researcher. `control` service assigns conditions randomly.
	control	All participants are provided exact same software and environment through Docker images.
	real-world	Programming environment based on Jupyter notebook/Visual Studio. Browser is Firefox instance.
	data collection	Data collected from participant in real-time through participant interface.
	ethics	Data stored on secure sever with pseudoanonymous identifier.
Technical	*isolation*	Participants use isolated network and file system within Docker containers. Resource limits prevent participants from using all the systems resources.
	scalability	Lightweight participant Docker containers created by `control` service as demand increases. No limit on number of Docker containers concurrently running.
	adaptability	Participant instance can be any web app packaged inside a Docker container.

Table 1: Each requirement and how NERDS satisfies it

directed to the system with a pseudoanonoymous identifier which is used to associate all participant data.

(2) The participant is shown an introduction and instructions for the study on the `landing` service. Asynchronously, the `landing` service gets a new participant instance from the ready pool.

(3) Once the participant instance is ready, the user is proxied to the instance software via the `nginx` service. The participant completes the study within the instance. All data from the instance is communicated to the backend through the `submit` service

(4) Once the participant finishes the study in the instance container, they are redirected to an external exit survey with their pseudoanonomous identifier and other associated information (e.g. the condition assigned). The system destroys the participant's instance as it is no longer needed.

A If instead the participant does not fully finish, the `control` service will destroy the instance after a pre-configured amount of time.

3.2.2 Participant Instances. The participant instances are responsible for providing the space and infrastructure for participants to complete the study. For this study, the instances were Docker containers with a highly customized version of Jupyter notebook, a web-based Python programming environment[2]. The web interface presents programming tasks defined by the study administrators to the participant in sequential order, hiding future tasks until they are reached. Participants have the option to run their code to test it, utilizing predefined tests we provide to them. Run output will appear below the code for review by the programmer. The participant can choose to either mark a task as complete or skip a task. See Figure 2 for a screenshot of the interface of the participant instance.

The participant instance records data about the participants progress throughout the study. All code runs are recorded and sent to the infrastructure with associated output, giving researchers and understanding of how participants progressed through the study. The instance also records how long the participant has the
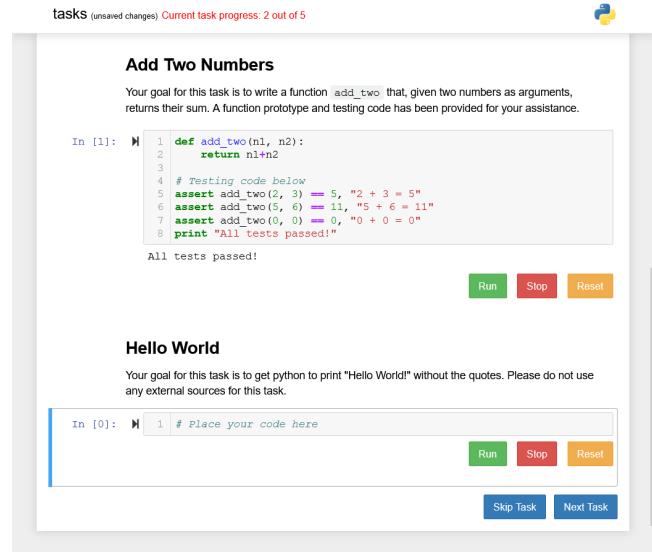

Figure 2: Screenshot of the participant coding environment. The environment is a heavily-modified Jupyter notebook to present programming tasks in order and report progress to the NERDS system.

window "focused,"[3], giving researchers insight into the time spent by participants focusing on the task.

As described in Section 3.1.3, participant instances must be isolated from each other and from the system itself (***isolation***). Full isolation is accomplished through Docker, as Docker infrastructure prevents access to file systems, networks, and resources in other containers. Participant code is run as an unprivileged user inside of the container preventing superuser access. Participant instances are given limited CPU resources through Docker, meaning the threat of a Denial-of-Service attack is minimal. Finally, participant instances are blocked from accessing the public internet.

---

[2]https://jupyter.org/

[3]Focused means having the browser tab with our system actively selected. This is reported by the browser through JavaScript.

In order to prevent the `manager` service in the study infrastructure from destroying an instance while a participant is still finishing the study (i.e. a participant taking advantage of the ***flexibility*** requirement), the instance sends periodic "heartbeats" to the backend while the user is using the interface. This allows the system to manage participant instances and shutdown any unused or abandoned instances after a researcher-determined amount of time.

3.2.3 Design advantages and limitations. Our fully containerized design has multiple advantages over the original Developer Observatory design [22]. Our system does not require using a specific hosting provider, unlike the previous system that required Amazon EC2. Any system capable of hosting Docker can host our system, which as of writing is Mac, Linux, and Windows. Since containers are less resource intensive, the overall computational cost of the system (and thus monetary cost) is substantially lower for a given number of participants making ***scalability*** more attainable. We also maintain the same security requirements (***isolation***) for far less computational cost.

One of the main limitations of the Developer Observatory was a limited number of concurrent participants and long wait times for new participants. These limitations are directly related to the use of Amazon EC2 instances to host the system. The limited number of concurrent participants was due to the hard limit of 50 running Amazon EC2 instances at a single time. The wait times come from the inherent increased cost of starting an entire operating system for every participant, which took about a minute in the Developer Observatory [22]. Our system addresses both of these limitations with containers. Since the participant instances are now hosted within containers, the number of concurrent participants is now unlimited, only bounded by the amount of resources available on the system. Containers also have little startup time, resulting in short wait times for participants. Both of these aspects drastically increase the ***scalability*** of the studies run on **NERDS**.

As the participant instances are Docker containers, they can also be easily switched out for any other participant interface as needed. The participant instance simply needs to be a Docker container with a web interface and periodically report participant data through documented calls. This means the **NERDS** system can be used for any study that needs separate, isolated instances for every participant. In the Developer Observatory, such seamless switching was not possible as the participant instances were EC2 templates and would need to be recreated for every instance of a study.

3.3 Deploying our system and lessons learned

Next, we detail our experience using **NERDS** for a large-scale secure software development study and the lessons we learned from this. This study was approved by University of Maryland's and Colorado School of Mines' ethical review boards.

3.3.1 Study overview. First, we give a brief overview of the study. As described in Section 3.1.4, the study required participants to *Write* code from scratch, *Read* existing code and identify issues (security and functionality), or *Fix* the issues (security and functionality) in existing code utilizing one of two Python libraries. Over a year long period, 141 participants started this study and 127 participants completed it. We saw substantially fewer dropouts than one of the original Developer Observatory studies [1]. We collected more than 5,500 code run events. Participants spent an average of 30.2 minutes participating in the study, including time spent on the final survey.

3.3.2 Utilizing the system. Next, we discuss our experience utilizing this system as researchers. The deployment process of **NERDS** was substantially easier than the Developer Observatory. For **NERDS**, we used a single server with 16 GB of RAM hosted on our University's compute cluster. While the containers could be hosted on separate systems, we opted to keep all on the same host for simplicity. In contrast, the Developer Observatory required creating an AWS account and a multi-step process to create AWS templates with the correct software installed.

The maintenance and updating of **NERDS** was also much easier than the original Developer Observatory. As new requirements or issues within the system arose, we could alter our system and rebuild the entire system with a single command. The next participant in the study (after the ready pool was exhausted) would automatically use the new instance with the fix applied. Additionally, building with Docker is incredibly fast, as Docker caches and reuses previous builds for components that have not changed. In contrast, the Developer Observatory requires altering each VM in the system anytime a change is made, which is a lengthy, highly manual process.

Running the study was straightforward and successful. Monitoring provided by the `stats` service gave quick insight into how many participants had successfully completed the study, providing key insights to researchers on condition balance. This made it very easy to determine whether any weighting was needed for condition assignment and whether we had received sufficient participants to conclude the study. Automated data exports from the database made analysis seamless as the data came pre-formatted (JSON). We received no reports of participants not being able to enter the study due to system load. No participants experienced interruption while participating in the study due to a system error or outage. The data was exported easily from the system's database and analyzed using separate tools by the researchers.

3.3.3 Lessons learned. We learned a few key lessons from conducting this study. First, participants will always find ways around system restrictions. In the study, participants in the *Read* condition were not allowed to run the code, but some participants discovered that they could still run the code using Jupyter's hotkeys. We removed data from these participants and quickly published a fix that disabled Jupyter's hotkey engine. Unfortunately, this also disabled legitimate uses of hotkeys for participants in the *Fix* and *Write* conditions. While striving for as much of a ***real-world*** participant experience as possible, there are trade-offs that have to be made.

Second, we learned that the information provided by the `stats` feature was invaluable. With a lab-study, monitoring progress is simple as researchers are actively involved in every participant interaction (a key trade-off of these studies). Conversely, with a remote study, closely monitoring participants becomes very difficult as participants are often working in tandem and on the study at a time that is most convenient for them (often including breaks). The `stats` feature allowed for us to closely monitor the study as it progressed. Specifically, the `stats` tool allowed us to discover a

large imbalance in the conditions assigned to participants where half as many participants were assigned to the *Write* condition compared to the other conditions. The stats tool allowed us to identify this and quickly push a fix to the landing service to weight the condition assignment.

Finally, while our participant instance collected a lot of data, there were some key insights that we lost due to the limitations of our system. While we could see how long it took a participant to complete the study and how long a participant spent "focused" on the study, we could not see what the participant was doing when they were not "focused". This was especially important, as we wanted participants to only use our study provided documentation rather than other internet resources. In a lab study, this would be trivial for researchers to see, but due to the limitations of a remote study, collecting such data is non-trivial. These trade-offs are something researchers need to consider in the study-design phase to ensure they control for all study-important factors.

Despite these limitations, conducting a remote study was far easier than an in-person study and promoted strong external and internal validity. Specifically, our participants were incredibly geographically, programming experience, and security experience diverse. This would not have likely been possible if we had conducted a lab-study. We never had to reserve a room at our institution, reimburse travel, or actively sit with participants as they completed our study. Our fully automated and remote approach meant we could collect far more data than would be possible in a lab study. While we did not get all of the information that we would have in a lab-study, we were still able to collect a substantial amount of data and maintain a enough control to ensure experimental validity.

4 TAKING NERDS FURTHER

With the success of the first study, we sought to further utilize **NERDS**. Specifically, we wanted to explore the ***adaptability*** of **NERDS**. Can we use **NERDS** for a completely different type of study? Can we change or add to the data collected during the study? We detail our unique requirements for the next iteration of **NERDS**, the design based on these requirements, the steps it took to rework **NERDS**, and our experience deploying the modified **NERDS**.

4.1 Requirements

In addition to the requirements detailed in Section 3.1, **NERDS** has additional study-specific requirements for the participant instances. With this second study, we wanted to have participants write code in the C programming language rather than in Python. Additionally, we wanted to be able to collect all participant search history to understand resources utilized during Study 2, given that this was a key piece of information we missed in Study 1. With these new requirements came two new challenges. First, running C code would require a substantial rework of the participant instance, as the previous study was limited to running Python inside of Jupyter Notebooks, which are unable to run C code. Second, collecting browser activity without requiring a participant to install any additional software beyond a browser itself (the ***non-invasive*** requirement of our system) is impossible due to the limitations of browser APIs.
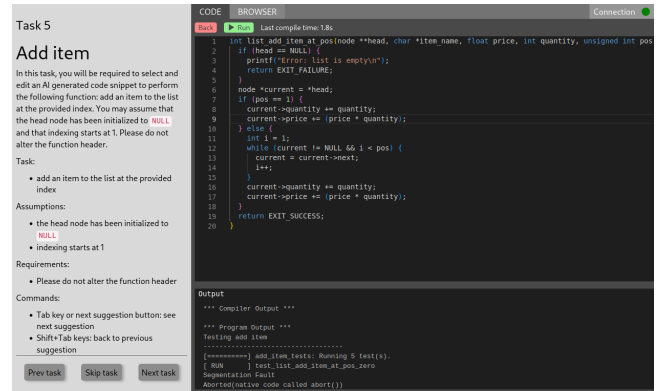

Figure 3: Screenshot of the C development environment. Task description is shown to the participant on the left. Code is edited on the right and output from the compiled WebAssembly is shown below the editor.

4.2 Design

Given these new challenges and requirements, we utilized the existing **NERDS** study infrastructure design described in Section 3.2.1. However, we needed to make heavy modifications to the participant instance to address our new study requirements. Our modified **NERDS** participant instance design has three main parts: (1) a development environment for the C language, (2) a system for compiling and running C code, and (3) an embedded browser for participants to use. See Figure 5 for a diagram of the participant instance.

4.2.1 C Development Environment. We developed a web-based C development environment for participants to use while completing the study. The environment is task-based, meaning it presents tasks in sequential order, allowing participants to submit a response for each. As one of the requirements is to mimic real development experience (***real-world***), we modeled the interface to resemble Visual Studio Code (VSCode), a popular IDE. We built the code editor using the same library[4] that VSCode uses, allowing our interface to mimic the real VSCode as closely as possible. See Figure 3 for a screenshot of the C development environment.

4.2.2 C Compiler. Instead of compiling and executing C code natively on the participant instance, which could introduce possible security concerns, we compile C code to WebAssembly and execute it on the participant's browser, promoting as much ***isolation*** as possible. WebAssembly is a binary format for executables designed to be executed on web browsers and is supported by most major browsers[5]. We use Emscripten[6] as the WebAssembly compiler on the participant instance. We compile participant code into a configurable testing suite that displays test results to the participant in a window below the editor when run. Running participant code on the participant's browser means any malicious code a participant uploads executes inside of their own browser instead of the study system (***isolation***). It also means the system has overall less load

---

[4]https://microsoft.github.io/monaco-editor/
[5]https://webassembly.org/
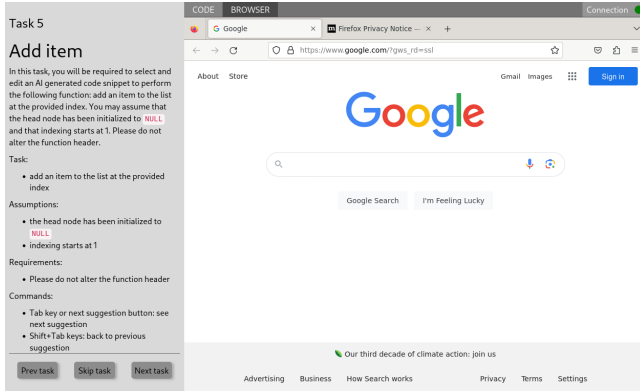[6]https://emscripten.org/

Figure 4: Screenshot of the embedded browser in the modified NERDS system. Browser history is collected automatically through a small script on the participant instance.

Figure 5: Diagram of redesigned participant instance for the second NERDS study.

as participants are using their own compute resources to run their code (*scalability*).

4.2.3 Embedded Browser. Since lacking internet search history was a key lesson learned from our first deployment of **NERDS** (Section 3.3.3), we wanted to collect any internet searches in our second deployment. To collect participant browser activity without requiring participants to install anything (*non-invasive*), we provided a full Firefox browser embedded in the participant interface. The Firefox process is hosted on the participant instance and made available through Virtual Network Computing (VNC), a remote desktop protocol. The web interface uses noVNC[7], a browser implementation of the VNC protocol, to display the embedded browser inside of the participant's browser. When the participant submits code to compile, a small Python script on the instance collects the hosted browser's history from the container's filesystem. In the instructions of our study we asked the participant to do all searches in the provided browser instead of using their own browser. See Figure 4 for a screenshot of the embedded browser.

4.2.4 Design advantages and limitations. The primary advantage of our new system is the ability to run arbitrary C code with associated tests without any security concerns. The C code runs on the participant's browser without any ability to affect the participant instance or the **NERDS** infrastructure (*isolation*). We can also collect participant browser data without any additional software installed on the participant's device (*non-invasive*).

The new system also demonstrates the relative ease of reworking the participant instance to work with a different language and collect different data (*adaptability*). While developing the instance itself incurred high development cost, the core **NERDS** infrastructure remains unchanged and supports the new participant instance in the exact same way. This could feasibly be done by other researchers for different study requirements.

There are a few limitations with this updated design. First, participant instances are now more resource intensive given they host an entire window server and a Firefox browser. This means the overall
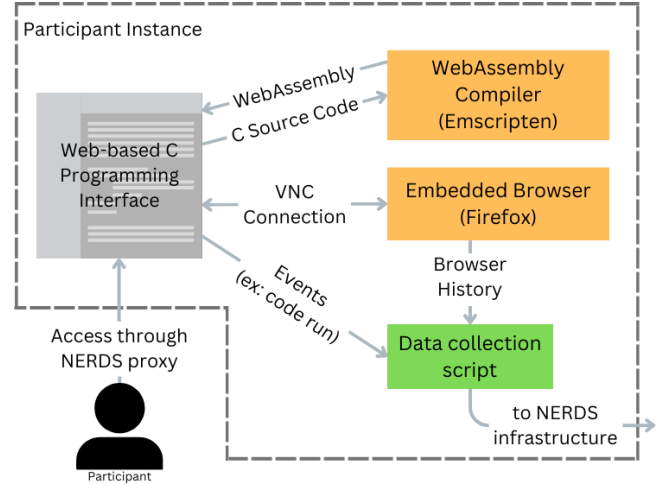
system resource requirements are higher for a given number of concurrent participants. While this slightly decreases *scalability*, the number of active participants is only bounded by the available resources. Acquiring more computational resources allows more participants to use the system concurrently.

Participants are also not forced to use the embedded browser, and may actively choose to avoid it. Due to limitations in the web environment, hotkeys such as `Ctrl-T` and the clipboard cannot be passed to the embedded browser. Power users, like programmers, might expect these features to be present (*real-world*). Additionally, participants with high network latency to the **NERDS** system may also experience frustration using the VNC session, as interacting with the embedded browser will be less responsive than natural interactions. These differences may cause participants to switch to their own browser. In such a cases, our system would not be able to collect browser data from the participant.

There are also difficulties with the WebAssembly-based execution model. WebAssembly may have significant differences in execution compared with a native environment. For example, a null dereference (access at memory location 0) is valid in WebAssembly since memory is implemented in a JavaScript array [9]. We ran into many difficulties with this as our test suite explicitly tested for null dereferences, which were not identified in the WebAssembly environment. This meant that running our provided tests natively and using WebAssembly produced different results. Our solution was to use many checks and sanitizers compiled into the final executable, such as `UBSan`, Clang's undefined behavior sanitizer. However, there may be some edge cases that `UBSan` does not detect.

4.3 Deploying our system and lessons learned

In this section, we detail the deployment of this modified **NERDS** system and the lessons we have learned so far. This study was approved by University of Maryland's and Colorado School of Mines' ethical review boards.

---

[7]https://novnc.com

4.3.1 Study overview. The study we redesigned **NERDS** for required participants to complete 4 tasks in the C programming language utilizing provided, completed code suggestions. Participants are provided multiple suggestion and directed to pick one. Once they select a suggestion, they are able to edit and test it. Each suggestion varies in its functionality and security. We deployed this redesigned system to conduct this study on April 8, 2024. As of writing, we have 25 finished participants and the data collection is ongoing. Participants have spent an average of 29.95 minutes completing the study. We have had no reports of interface issues or system outages.

4.3.2 Utilizing the system. Developing the new participant instance took a substantial amount of work, largely due to the complexity of hosting the embedded browser and designing a brand new development environment. However, as described in Section 4.2.4, the process of switching out the old Jupyter-based instance with the new instance (once it was completed) was seamless. The only changes that were made to the **NERDS** study infrastructure were fixes for small bugs discovered during the development of the new participant instance.

While we anticipated that the participant instances required more resources than in Study 1 (see Section 4.2.4), we hosted Study 2 on the same system used to host Study 1. We experimented with the increased resource requirements by performing a stress test of the system before we launched the study. We specifically wanted to see how many people could be concurrently using their embedded browsers without impacting the experience of other users. Rough results showed around 10 people could perform intensive tasks like watching videos inside the browser and without having a noticeable affect on the experience of other participants. However, 10 participants is not an upper bound, and we would need to further explore the hard upper limit for **NERDS**. We deemed this soft upper bound sufficient, as we expected most participants would not be doing extremely intensive tasks within their browser, and decided to launch the study with the same amount of resources as Study 1. We have not received any reports of degraded performance from participants thus far.

The preliminary data is rich, but as anticipated in section 4.2.4, it seems only some participants are using the embedded browser. As of writing, only 2 participants used the browser at all. Given the nature of the study, this is not entirely surprising as participants are given completed code, so they may not need online resources. However, it is also possible that they used their own browser to look up information. If this number does not improve throughout the course of our study we may explore alternative methods for collecting browser information in future studies.

4.3.3 Lessons learned. As of writing the study has been running for around 3 months, so our lessons learned are preliminary, and we expect to learn more as the study continues.

During the first few weeks of the study, one participant was kicked out of the system while completing the study due to an unintended command run by a developer of the system. While the participant was eventually able to recover and resume the study, better checks are needed to prevent editing the system internals while participants are actively completing a study. For our wider release to the research community, such edits while the study is running should be very hard, if not impossible, to make.

The relative ease of switching out the participant instance between Study 1 and Study 2 showed us that this system is very extensible and adaptable for a wide range of studies. With documentation, this system can be useful to any researcher wishing to run a remote study that needs isolated instances for participants.

5 FUTURE WORK

While **NERDS** is a valuable system for conducting remote developer studies, there are many improvements that could be made. First, while we are making this system publicly available to the research community, the system may be hard to use for researchers. Clear documentation and an easy-to-use study management interface should be created to assist in creating, running, and collecting data from a study. Additionally, creating the new participant instance for Study 2 required a high amount of overhead and developer experience on the part of the study administrators. Thus, creating custom instances may be difficult or impossible for some researchers. More experience is needed in creating custom participant instances in various studies. Potentially, a library of instances for various study types could be made available for researchers for a more "plug-and-play" experience. While our experience was relatively straightforward, we would love to hear from other researchers if they find the experience intuitive enough to use for their own study.

Second, the system architecture is heavily coupled to Docker. If the system could run on any container system, this would further improve the ***adaptability*** of **NERDS**. For example, the popular container orchestration Kubernettes may be able to run our system more efficiently and at a greater scale than Docker.

Finally, the embedded browser in the participant instance of Study 2 is a weak point of our system currently, as it limits the ***real-world*** likeness of **NERDS**. Adding the ability to pass clipboard data and hotkeys between the embedded browser and the participant's system would vastly improve the experience for developers participating in the study.

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the usability of cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 154–171.

[2] Yasemin Acar, Sascha Fahl, and Michelle L Mazurek. 2016. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 3–8.

[3] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. 2017. Security developer studies with Github users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. 81–95.

[4] Sebastian Baltes and Stephan Diehl. 2016. Worse than spam: Issues in sampling software developers. In *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*. 1–6.

[5] Gunnar R Bergersen, Dag IK Sjøberg, and Tore Dybå. 2014. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering* 40, 12 (2014), 1163–1184.

[6] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. 2011. Trend analysis of the cve for software vulnerability management. In *Proceedings of the 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. IEEE, 1290–1293.

[7] Anastasia Danilova, Stefan Horstmann, Matthew Smith, and Alena Naiakshina. 2022. Testing time limits in screener questions for online surveys with programmers. In *Proceedings of the 44th International Conference on Software Engineering*. 2080–2090.

[8] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. 2021. Do you really code? Designing and Evaluating Screening Questions for Online Surveys with Programmers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 537–548.

[9] Emscripten. [n. d.]. Debugging with Sanitizers. https://emscripten.org/docs/debugging/Sanitizers.html?highlight=dereference#catching-null-dereference|.

[10] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE, 73–82.

[11] Kelsey R Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L Mazurek, Michael Hicks, and James Parker. 2022. Understanding the how and the why: Exploring secure development practices through a course competition. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1141–1155.

[12] Nicolas Huaman, Alexander Krause, Dominik Wermke, Jan H. Klemmer, Christian Stransky, Yasemin Acar, and Sascha Fahl. 2022. If You Can't Get Them to the Lab: Evaluating a Virtual Study Environment with Security Information Workers. 313–330.

[13] Harjot Kaur, Sabrina Amft, Daniel Votipka, Yasemin Acar, and Sascha Fahl. 2022. Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples. (2022).

[14] Mitre. 2020. CVE. https://cve.mitre.org/.

[15] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. 2020. On conducting security developer studies with CS students: Examining a password-storage study with CS students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[16] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel Von Zezschwitz, and Matthew Smith. 2019. "If you want, I can store the encrypted password" A Password-Storage Field Study with Freelance Developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[17] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why do developers get password storage wrong? A qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 311–328.

[18] NIST. 2020. National Vulnerability Database. https://nvd.nist.gov/general.

[19] Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. 2017. Developer-centered security and the symmetry of ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop*. 46–56.

[20] 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*. 690–703.

[21] Raphael Serafini, Marco Gutfleisch, Stefan Albert Horstmann, and Alena Naiakshina. 2023. On the Recruitment of Company Developers for Security Studies: Results from a Qualitative Interview Study. In *Nineteenth Symposium on Usable Privacy and Security (SOUPS 2023)*. 321–340.

[22] Christian Stransky, Yasemin Acar, Duc Cuong Nguyen, Dominik Wermke, Doowon Kim, Elissa M. Redmiles, Michael Backes, Simson Garfinkel, Michelle L. Mazurek, and Sascha Fahl. 2017. Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/cset17/workshop-program/presentation/stransky

[23] Mohammad Tahaei and Kami Vaniea. 2022. Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List. In *CHI Conference on Human Factors in Computing Systems*. 1–15.

[24] Daniel Votipka, Desiree Abrokwa, and Michelle L Mazurek. 2020. Building and Validating a Scale for Secure Software Development Self-Efficacy. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–20.

[25] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 109–126.

[26] Aiko Yamashita and Leon Moonen. 2013. Surveying developer knowledge and interest in code smells through online freelance marketplaces. In *2013 2nd International Workshop on User Evaluations for Software Engineering Researchers (USER)*. IEEE, 5–8.