Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It

Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks
University of Maryland
{dvotipka,kfulton,jprider,mhou1,mmazurek,mwh}@cs.umd.edu

Abstract
Secure software development is a challenging task requiring consideration of many possible threats and mitigations. This paper investigates how and why programmers, despite a baseline of security experience, make security-relevant errors. To do this, we conducted an in-depth analysis of 94 submissions to a secure-programming contest designed to mimic real-world constraints: correctness, performance, and security. In addition to writing secure code, participants were asked to search for vulnerabilities in other teams’ programs; in total, teams submitted 866 exploits against the submissions we considered. Over an intensive six-month period, we used iterative open coding to manually, but systematically, characterize each submitted project and vulnerability (including vulnerabilities we identified ourselves). We labeled vulnerabilities by type, attacker control allowed, and ease of exploitation, and projects according to security implementation strategy. Several patterns emerged. For example, simple mistakes were least common: only 21% of projects introduced such an error. Conversely, vulnerabilities arising from a misunderstanding of security concepts were significantly more common, appearing in 78% of projects. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

1 Introduction
Developing secure software is a challenging task, as evidenced by the fact that vulnerabilities are still discovered, with regularity, in production code [21, 22, 54]. How can we improve this situation? There are many steps we could take. We could invest more in automated vulnerability discovery tools [7, 11, 12, 26, 49, 67, 72, 75, 76]. We could expand security education [19, 40, 42, 47, 59]. We could focus on improving secure development processes [20, 48, 53, 65].

An important question is which intervention is ultimately most effective in maximizing outcomes while minimizing time and other resources expended. The increasing pervasiveness of computing and the rising number of professional developers [18, 44, 77] is evidence of the intense pressure to produce new services and software quickly and efficiently. As such, we must be careful to choose interventions that work best in the limited time they are allotted. To do this, we must understand the general type, attacker control allowed, and ease of exploitation of different software vulnerabilities, and the reasons that developers make them. That way, we can examine how different approaches address the landscape of vulnerabilities.

This paper presents a systematic, in-depth examination (using best practices developed for qualitative assessments) of vulnerabilities present in software projects. In particular, we looked at 94 project submissions to the Build it, Break it, Fix it (BIBIFI) secure-coding competition series [66]. In each competition, participating teams (many of which were enrolled in a series of online security courses [2]) first developed programs for either a secure event-logging system, a secure communication system simulating an ATM and a bank, or a scriptable key-value store with role-based access control policies. Teams then attempted to exploit the project submissions of other teams. Scoring aimed to match real-world development constraints: teams were scored based on their project’s performance, its feature set (above a minimum baseline), and its ultimate resilience to attack. Our six-month examination considered each project’s code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

The BIBIFI competition provides a unique and valuable vantage point for examining the vulnerability landscape. When looking for trends in open-source projects, there are confounding factors: Different projects do different things, and were developed under different circumstances, e.g., with different resources and levels of attention. By contrast, in BIBIFI we have many implementations of the same problem carried out by different teams but under similar circumstances. As such, we can postulate the reasons for observed differences with more confidence. While less controlled than a lab study, BIBIFI offers more ecological validity—teams had weeks to build their project submissions, not days, using any languages,
tools, or processes they preferred.

Our rigorous manual analysis of this dataset both identified new insights about secure development and confirmed findings from lab studies and field measurements, all with implications for improving secure-development training, security-relevant APIs [4, 36, 57], and tools for vulnerability discovery.

Simple mistakes, in which the developer attempts a valid security practice but makes a minor programming error, were least common: only 21% of projects introduced such an error. Mitigations to these types of mistakes are plentiful. For example, in our data, minimizing the trusted code base (e.g., by avoiding duplication of security-critical code) led to significantly fewer mistakes. Moreover, we believe that modern analysis tools and testing techniques [8, 9, 15, 16, 25, 29, 38, 41, 43, 70, 71, 81] should uncover many of them. All but one of the mistakes in our dataset were found and exploited by opposing teams. In short, this type of bug appears to be both relatively uncommon and amenable to existing tools and best practices, suggesting it can be effectively managed.

On the other hand, vulnerabilities arising from misunderstanding of security concepts were significantly more common: 78% of projects introduced at least one such error. In examining these errors, we identify an important distinction between intuitive and unintuitive security requirements; for example, several teams used encryption to protect confidentiality but failed to also protect integrity. In 45% of projects, teams missed unintuitive requirements altogether, failing to even attempt to implement them. Teams also often made conceptual errors in attempting to apply a security mechanism (44% of projects); for example, several projects failed to provide randomness when an API expects it. Although common, these vulnerabilities proved harder to exploit: only 71% were exploited by other teams (compared to 97% of simple mistakes), and our qualitative labeling identified 35% as difficult to exploit (compared to none of the simple mistakes). These more complex errors expose a need for APIs less subject to misuse, better documentation, and better security training that focuses on less-intuitive concepts like integrity.

Overall, our findings suggest rethinking strategies to prevent and detect vulnerabilities, with more emphasis on conceptual difficulties rather than mistakes.

# Data

This section presents the Build It, Break It, Fix It (BIBIFI) secure-programming competition [66], the data we gathered from it which forms the basis of our analysis, and reasons why the data may (or may not) represent real-world situations.

## Build it, Break it, Fix it

A BIBIFI competition comprises three phases: building, breaking, and fixing. Participating teams can win prizes in both build-it and break-it categories.

In the first (build it) phase, teams are given just under two weeks to build a project that (securely) meets a given specification. During this phase, a team’s build-it score is determined by the correctness and efficiency of their project, assessed by test cases provided by the contest organizers. All projects must meet a core set of functionality requirements, but they may optionally implement additional features for more points. Submitted projects may be written in any programming language and are free to use open-source libraries, so long as they can be built on a standard Ubuntu Linux VM.

In the second (break it) phase, teams are given access to the source code of their fellow competitors’ projects in order to look for vulnerabilities. Once a team identifies a vulnerability, they create a test case (a break) that provides evidence of exploitation. Depending on the contest problem, breaks are validated in different ways. One is to compare the output of the break on the target project against that of a “known correct” reference implementation (RI) written by the competition organizers. Another way is by confirming knowledge (or corruption) of sensitive data (produced by the contest organizers) that should have been protected by the target project’s implementation. Successful breaks add to a team’s break-it score, and reduce the target project’s team’s build-it score.

The final (fix it) phase of the contest affords teams the opportunity to fix bugs in their implementation related to submitted breaks. Doing so has the potential benefit that breaks which are superficially different may be unified by a fix, preventing them from being double counted when scoring.

## 2.2 Data gathered

We analyzed projects developed by teams participating in four BIBIFI competitions, covering three different programming problems: secure log, secure communication, and multiuser database. (Appendix A provides additional details about the makeup of each competition.) Each problem specification required the teams to consider different security challenges and attacker models. Here we describe each problem, the size/makeup of the reference implementation (for context), and the manner in which breaks were submitted.

Secure log (SL, Fall 2014 and Spring 2015, RI size: 1,013 lines of OCaml). This problem asks teams to implement two programs: one to securely append records to a log, and one to query the log’s contents. The build-it score is measured by log query/append latency and space utilization, and teams may implement several optional features.

Teams should protect against a malicious adversary with access to the log and the ability to modify it. The adversary does not have access to the keys used to create the log. Teams are expected (but not told explicitly) to utilize cryptographic functions to both encrypt the log and protect its integrity.

---

1 Source code obfuscation was against the rules. Complaints of violations were judged by contest organizers.
During the break-it phase, the organizers generate sample logs for each project. Break-it teams demonstrate compromises to either integrity or confidentiality by manipulating a sample log file to return a differing output or by revealing secret content of a log file.

Secure communication (SC, Fall 2015, RI size: 1,124 lines of Haskell). This problem asks teams to build a pair of client/server programs. These represent a bank and an ATM, which initiates account transactions (e.g., account creation, deposits, withdrawals, etc.). Build-it performance is measured by transaction latency. There are no optional features.

Teams should protect bank data integrity and confidentiality against an adversary acting as a man-in-the-middle (MITM), with the ability to read and manipulate communications between the client and server. Once again, build teams were expected to use cryptographic functions, and to consider challenges such as replay attacks and side-channels. Break-it teams demonstrate exploitations violating confidentiality or integrity of bank data by providing a custom MITM and a script of interactions. Confidentiality violations reveal the secret balance of accounts, while integrity violations manipulate the balance of unauthorized accounts.

Multiuser database (MD, Fall 2016, RI size: 1,080 lines of OCaml). This problem asks teams to create a server that maintains a secure key-value store. Clients submit scripts written in a domain-specific language. A script authenticates with the server and then submits a series of commands to read/write data stored there. Data is protected by role-based access control policies customizable by the data owner, who may (transitively) delegate access control decisions to other principals. Build-it performance is assessed by script running time. Optional features take the form of additional script commands.

The problem assumes that an attacker can submit commands to the server, but not snoop on communications. Break-it teams demonstrate vulnerabilities with a script that shows a security-relevant deviation from the behavior of the RI. For example, a target implementation has a confidentiality violation if it returns secret information when the RI denies access.

Project Characteristics. Teams used a variety of languages in their projects. Python was most popular overall (39 teams, 41%), with Java also widely used (19, 20%), and C/C++ third (7 each, 7%). Other languages used by at least one team include Ruby, Perl, Go, Haskell, Scala, PHP, JavaScript Visual Basic, OCaml, C#, and F#. For the secure log problem, projects ranged from 149 to 3857 lines of code (median 1095), secure communication ranged from 355 to 4466 (median 683) and multiuser database from 775 to 5998 (median 1485).

2.3 Representativeness: In Favor and Against

Our hope is that the vulnerability particulars and overall trends that we find in BIBIFI data are, at some level, representative of the particulars and trends we might find in real-world code. There are several reasons in favor of this view:

- Scoring incentives match those in the real world. At build-time, scoring favors features and performance—security is known to be important, but is not (yet) a direct concern. Limited time and resources force a choice between uncertain benefit later or certain benefit now. Such time pressures mimic short release deadlines.

- The projects are substantial, and partially open ended, as in the real world. For all three problems, there is a significant amount to do, and a fair amount of freedom about how to do it. Teams must think carefully about how to design their project to meet the security requirements. All three projects consider data security, which is a general concern, and suggest or require general mechanisms, including cryptography and access control. Teams were free to choose the programming language and libraries they thought would be most successful. While real-world projects are surely much bigger, the BIBIFI projects are big enough that they can stand in for a component of a larger project, and thus present a representative programming challenge for the time given.

- About three-quarters of the teams whose projects we evaluated participated in the contest as the capstone to an on-line course sequence (MOOC) [2]. Two courses in this sequence — software security and cryptography — were directly relevant to contest problems. Although these participants were students, most were also post-degree professionals; overall, participants had a average of 8.9 years software development experience. Further, prior work suggests that in at least some secure development studies, students can substitute effectively for professionals, as only security experience, not general development experience, is correlated with security outcomes [5, 6, 56, 58].

On the other hand, there are several reasons to think the BIBIFI data will not represent the real world:

- Time pressures and other factors may be insufficiently realistic. For example, while there was no limit on team size (they ranged from 1 to 7 people with a median of 2), some teams might have been too small, or had too little free time, to devote enough energy to the project. That said, the incentive to succeed in the contest in order to pass the course for the MOOC students was high, as they would not receive a diploma for the whole sequence otherwise. For non-MOOC students, prizes were substantial, e.g., $4000 for first prize. While this may not match the incentive in some security-mature companies where security is “part of the job” [37] and continued employment rests on good security practices, prior work suggests that many companies are not security-mature [10].

- We only examine three secure-development scenarios.
These problems involve common security goals and mechanisms, but results may not generalize outside them to other security-critical tasks.

- BIBIFI does not simulate all realistic development settings. For example, in some larger companies, developers are supported by teams of security experts [78] who provide design suggestions and set requirements, whereas BIBIFI participants carry out the entire development task. BIBIFI participants choose the programming language and libraries to use, whereas at a company the developers may have these choices made for them. BIBIFI participants are focused on building a working software package from scratch, whereas developers at companies are often tasked with modifying, deploying, and maintaining existing software or services. These differences are worthy of further study on their own. Nevertheless, we feel that the baseline of studying mistakes made by developers tasked with the development of a full (but small) piece of software is an interesting one, and may indeed support or inform alternative approaches such as these.

- To allow automated break scoring, teams must submit exploits to prove the existence of vulnerabilities. This can be a costly process for some vulnerabilities that require complex timing attacks or brute force. This likely biases the exploits identified by breaker teams. To address this issue, two researchers performed a manual review of each project to identify and record any hard to exploit vulnerabilities.

- Finally, because teams were primed by the competition to consider security, they are perhaps more likely to try to design and implement their code securely [57, 58]. While this does not necessarily give us an accurate picture of developer behaviors in the real world, it does mirror situations where developers are motivated to consider security, e.g., by security experts in larger companies, and it allows us to identify mistakes made even by such developers.

Ultimately, the best way to see to what extent the BIBIFI data represents the situation in the real world is to assess the connection empirically, e.g., through direct observations of real-world development processes, and through assessment of empirical data, e.g., (internal or external) bug trackers or vulnerability databases. This paper’s results makes such an assessment possible: Our characterization of the BIBIFI data can be a basis of future comparisons to real-world scenarios.

3 Qualitative Coding

We are interested in characterizing the vulnerabilities developers introduce when writing programs with security requirements. In particular, we pose the following research questions:

RQ1 What types of vulnerabilities do developers introduce? Are they conceptual flaws in their understanding of security requirements or coding mistakes?

RQ2 How much control does an attacker gain by exploiting the vulnerabilities, and what is the effect?

RQ3 How exploitable are the vulnerabilities? What level of insight is required and how much work is necessary?

Answers to these questions can provide guidance about which interventions—tools, policy, and education—might be (most) effective, and how they should be prioritized. To obtain answers, we manually examined 94 BIBIFI projects (67% of the total), the 866 breaks submitted during the competition, and the 42 additional vulnerabilities identified by the researchers through manual review. We performed a rigorous iterative open coding [74, pg. 101-122] of each project and introduced vulnerability. Iterative open coding is a systematic method, with origins in qualitative social-science research, for producing consistent, reliable labels (‘codes’) for key concepts in unstructured data. The collection of labels is called a codebook. The ultimate codebook we developed provides labels for vulnerabilities—their type, attacker control, and exploitability—and for features of the programs that contained them.

This section begins by describing the codebook itself, then describes how we produced it. An analysis of the coded data is presented in the next section.

3.1 Codebook

Both projects and vulnerabilities are characterized by several labels. We refer to these labels as variables and their possible values as levels.

3.1.1 Vulnerability codebook

To measure the types of vulnerabilities in each project, we characterized them across four variables: Type, Attacker Control, Discovery Difficulty, and Exploit Difficulty. The structure of our vulnerability codebook is given in Table 1. Our coding scheme is adapted in part from the CVSS system for scoring vulnerabilities [32]. In particular, Attacker Control and Exploit Difficulty relate to the CVSS concepts of Impact, Attack Complexity, and Exploitability. We do not use CVSS directly, in part because some CVSS categories are irrelevant to our dataset (e.g., none of the contest problems involve human interactions). Further, we followed qualitative best practices of letting natural (anti)patterns emerge from the data, modifying the categorizations we apply accordingly.

Vulnerability type. The Type variable characterizes the vulnerability’s underlying source (RQ1). For example, a vulnerability in which encryption initialization vectors (IVs) are reused is classified as having the issue insufficient randomness. The underlying source of this issue is a conceptual misunderstanding of how to implement a security concept.

Hence, our use of the term “coding” refers to a type of structured categorization for data analysis, not a synonym for programming.

The last column indicates Krippendorff’s α statistic [39], which we discuss in Section 3.2.
We identified more than 20 different issues grouped into three types; these are discussed in detail in Section 4.

**Attacker control.** The *Attacker Control* variable characterizes the impact of a vulnerability’s exploitation (RQ2) as either a full compromise of the targeted data or a partial one. For example, a secure-communication vulnerability in which an attacker can corrupt any message without detection would be a full compromise, while only being able to corrupt some bits in the initial transmission would be coded as partial.

**Exploitability.** We indicated the difficulty to produce an exploit (RQ3) using two variables, *Discovery Difficulty* and *Exploit Difficulty*. The first characterizes the amount of knowledge the attacker must have to initially find the vulnerability. There are three possible levels: only needing to observe the project’s inputs and outputs (*Execution*); needing to view the project’s source code (*Source*); or needing to understand key algorithmic concepts (*Deep Insight*). For example, in the secure-log problem, a project that simply stored all events in a plaintext file with no encryption would be coded as *Execution* since neither source code nor deep insight would be required for exploitation. The second variable, *Exploit Difficulty*, describes the amount of work needed to exploit the vulnerability once discovered. This variable has four possible levels of increasing difficulty depending on the number of steps required: only a single step, a small deterministic set of steps, a large deterministic set of steps, or a large probabilistic set of steps. As an example, in the secure-communication problem, if encrypted packet lengths for failure messages are predictable and different from successes, this introduces an information leakage exploitable over multiple probabilistic steps. The attacker can use a binary search to identify the initial deposited amount by requesting withdrawals of varying values and observing which succeed.

### Table 1: Summary of the vulnerability codebook.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Levels</th>
<th>Description</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>(See Table 2)</td>
<td>What caused the vulnerability to be introduced</td>
<td>0.85, 0.82</td>
</tr>
<tr>
<td>Attacker Control</td>
<td>Full / Partial</td>
<td>What amount of the data is impacted by an exploit</td>
<td>0.82</td>
</tr>
<tr>
<td>Discovery Difficulty</td>
<td>Execution / Source / Deep Insight</td>
<td>What level of sophistication would an attacker need to find the vulnerability</td>
<td>0.80</td>
</tr>
<tr>
<td>Exploit Difficulty</td>
<td>Single step / Few steps / Many steps / Probabilistic</td>
<td>How hard it would be for an attacker to exploit the vulnerability once discovered</td>
<td>1</td>
</tr>
</tbody>
</table>

3.2 Coding Process

Now we turn our attention to the process we used to develop the codebook just described. Our process had two steps: Selecting a set of projects for analysis, and iteratively developing a codebook by examining those projects.

3.2.1 Project Selection

We started with 142 qualifying projects in total, drawn from four competitions involving the three problems. Manually analyzing every project would be too time consuming, so we decided to consider a sample of 94 projects—just under 67% of the total. We did not sample them randomly, for two reasons. First, the numbers of projects implementing each problem are unbalanced; e.g., secure log comprises just over 50% of the total. Second, a substantial number of projects had no break submitted against them—57 in total (or 40%). A purely random sample from the 142 could lead us to considering too many (or too few) projects without breaks, or too many from a particular problem category.

To address these issues, our sampling procedure worked as follows. First, we bucketed projects by the problem solved, and sampled from each bucket separately. This ensured that we had roughly 67% of the total projects for each problem. Second, for each bucket, we separated projects with a submitted break from those without one, and sampled 67% of the projects from each. This ensured we maintained the relative break/non-break ratio of the overall project set. Lastly, within the group of projects with a break, we divided them into four equally-sized quartiles based on number of breaks found dur-
To develop our codebooks, two researchers first cooperatively analyzed vulnerabilities to establish the initial codebook. This process continued until a reasonable level of inter-rater reliability was reached for each variable. Inter-rater reliability measures the agreement or consensus between different researchers applying the same codebook. To measure inter-rater reliability, we used the Krippendorff’s α statistic [39]. Krippendorff’s α is a conservative measure which considers improvement over simply guessing. Krippendorff et al. recommend a threshold of $\alpha > 0.8$ as a sufficient level of agreement [39]. The final Krippendorff’s α for each variable is given in Table 1. Because the Types observed in the MD problem were very different from the other two problems (e.g., cryptography vs. access control related), we calculated inter-rater reliability separately for this problem to ensure reliability was maintained in this different data. Once a reliable codebook was established, the remaining 34 projects (with 166 associated breaks) were divided evenly among the two researchers and coded separately.

Overall, this process took approximately six months of consistent effort by two researchers.

4 Vulnerability Types

Our manual analysis of 94 BIBIFI projects identified 182 unique vulnerabilities. We categorized each based on our codebook into 23 different issues. Table 2 presents this data. Issues are organized according to three main types: No Implementation, Misunderstanding, and Mistake (RQ1). These were determined systematically using axial coding, which identifies connections between codes and extracts higher-level themes [74, pg. 123-142]. For each issue type, the table gives both the number of vulnerabilities and the number of projects that included a vulnerability of that type. A dash indicates that a vulnerability does not apply to a problem.

This section presents descriptions and examples for each type. When presenting examples, we identify particular projects using a shortened version of the problem and a randomly assigned ID. In the next section, we consider trends in this data, specifically involving vulnerability type prevalence, attacker control, and exploitability.

4.1 No Implementation

We coded a vulnerability type as No Implementation when a team failed to even attempt to implement a necessary security mechanism. Presumably, they did not realize it was needed. This type is further divided into the sub-type All Intuitive, Some Intuitive, and Unintuitive. In the first two sub-types teams did not implement all or some, respectively, of the requirements that were either directly mentioned in the problem specification or were intuitive (e.g., the need for encryption to provide confidentiality). The Unintuitive sub-type was used if the security requirement was not directly stated or was otherwise unintuitive (e.g., using MAC to provide integrity [3]).

Two issues were typed as All Intuitive: not using encryption in the secure log ($P=3, V=3$) and secure communication ($P=2, V=2$) problems and not performing any of the specified access control checks in the multiuser database problem ($P=0, V=0$). The Some Intuitive sub-type was used when teams did not implement some of the nine multiuser database problem access-control checks ($P=10, V=18$). For example, several teams failed to check authorization for commands only $\text{admin}$ should be able to issue For Unintuitive vulnerabilities, there were four issues: teams failed to include a MAC to protect data integrity in the secure log ($P=16, V=16$) and secure communication ($P=7, V=7$) problems; prevent side-channel communication...
## Table 2: Number of vulnerabilities for each issue and the number of projects each vulnerability was introduced in.

<table>
<thead>
<tr>
<th>Type</th>
<th>Sub-type</th>
<th>Issue</th>
<th>Secure log</th>
<th>Secure communication</th>
<th>Multiuser database</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>P=52²</td>
<td>V=53²</td>
<td>P=15</td>
<td>P=94</td>
</tr>
<tr>
<td>No Impl. All Intuitive</td>
<td>No encryption</td>
<td>3 (6%)</td>
<td>2 (7%)</td>
<td>2 (3%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>No access control</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>–</td>
<td>–</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>3 (6%)</td>
<td>3 (6%)</td>
<td>2 (7%)</td>
<td>2 (3%)</td>
<td>–</td>
</tr>
<tr>
<td>Some Intuitive</td>
<td>Missing some access control</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>10 (67%)</td>
</tr>
<tr>
<td>Unintuitive</td>
<td>No MAC</td>
<td>16 (31%)</td>
<td>7 (26%)</td>
<td>7 (11%)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Side-channel attack</td>
<td>–</td>
<td>–</td>
<td>11 (41%)</td>
<td>11 (17%)</td>
<td>4 (15%)</td>
</tr>
<tr>
<td></td>
<td>No replay check</td>
<td>–</td>
<td>–</td>
<td>7 (26%)</td>
<td>7 (11%)</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>No recursive delegation check</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>4 (27%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>16 (31%)</td>
<td>16 (30%)</td>
<td>18 (67%)</td>
<td>25 (39%)</td>
<td>8 (53%)</td>
</tr>
<tr>
<td>Misund. Bad Choice</td>
<td>Unkeyed function</td>
<td>6 (12%)</td>
<td>6 (11%)</td>
<td>2 (7%)</td>
<td>2 (3%)</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Weak crypto</td>
<td>4 (8%)</td>
<td>5 (9%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Homemade crypto</td>
<td>2 (4%)</td>
<td>2 (4%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Weak AC design</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>5 (33%)</td>
</tr>
<tr>
<td></td>
<td>Memory corruption</td>
<td>1 (2%)</td>
<td>1 (2%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>13 (25%)</td>
<td>14 (26%)</td>
<td>2 (7%)</td>
<td>2 (3%)</td>
<td>5 (33%)</td>
</tr>
<tr>
<td>Conceptual Error</td>
<td>Fixed value</td>
<td>12 (23%)</td>
<td>12 (23%)</td>
<td>6 (22%)</td>
<td>6 (9%)</td>
<td>8 (53%)</td>
</tr>
<tr>
<td></td>
<td>Insufficient</td>
<td>2 (4%)</td>
<td>3 (6%)</td>
<td>5 (19%)</td>
<td>5 (8%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td></td>
<td>randomness</td>
<td>3 (6%)</td>
<td>3 (6%)</td>
<td>6 (22%)</td>
<td>7 (11%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td></td>
<td>Security on subset of data</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>1 (4%)</td>
<td>1 (2%)</td>
<td>2 (13%)</td>
</tr>
<tr>
<td></td>
<td>Library cannot handle input</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>1 (7%)</td>
</tr>
<tr>
<td>Resource exhaustion</td>
<td>0 (0%)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>17 (33%)</td>
<td>19 (36%)</td>
<td>15 (56%)</td>
<td>19 (30%)</td>
<td>9 (60%)</td>
</tr>
<tr>
<td>Mistake</td>
<td>Insufficient error checking</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>8 (30%)</td>
<td>8 (12%)</td>
<td>4 (27%)</td>
</tr>
<tr>
<td></td>
<td>Uncought runtime error</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>1 (4%)</td>
<td>1 (2%)</td>
<td>4 (27%)</td>
</tr>
<tr>
<td></td>
<td>Control flow mistake</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>1 (4%)</td>
<td>1 (2%)</td>
<td>4 (27%)</td>
</tr>
<tr>
<td></td>
<td>Skipped algorithmic step</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>4 (15%)</td>
<td>6 (9%)</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>Null write</td>
<td>1 (2%)</td>
<td>1 (2%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>1 (2%)</td>
<td>1 (2%)</td>
<td>11 (41%)</td>
<td>16 (25%)</td>
<td>8 (53%)</td>
</tr>
</tbody>
</table>

1 Number of projects submitted to the competition
2 Number of unique vulnerabilities introduced
3 Total percentages are based on the counts of applicable projects
data leakage through packet sizes or success/failure responses in the secure communication (P=11, V=11) and multiuser database (P=4, V=4) problems, respectively; prevent replay attacks (P=7, V=7) in the secure communication problem; and check the chain of rights delegation (P=4, V=4) in the multiuser database problem.

4.2 Misunderstanding

A vulnerability type was coded as Misunderstanding when a team attempted to implement a security mechanism, but failed due to a conceptual misunderstanding. We sub-typed these as either Bad Choice or Conceptual Error.

4.2.1 Bad Choice

Five issues fall under this sub-type, which categorizes algorithmic choices that are inherently insecure.

The first three issues relate to the incorrect implementation of encryption and/or integrity checks in the SL and SC problems: use of an algorithm without any secret component, i.e., a key (P=8, V=8), weak algorithms (P=4, V=5), or homemade encryption (P=2, V=2). As an example of a weak algorithm, SL-69 simply XOR’d key-length chunks of the text with the user-provided key to generate the final ciphertext. Therefore, the attacker could simply extract two key-length chunks of the ciphertext, XOR them together and produce the key.

The next issue identifies a weak access-control design for the MD problem, which could not handle all use cases (P=5, V=6). For example, MD-14 implemented delegation improperly. In the MD problem, a default delegator may be set by the administrator, and new users should receive the rights this delegator has when they are created. However, MD-14 granted rights not when a user was created, but when they accessed particular data. If the default delegator received access to data between time of the user’s creation and time of access, the user would be incorrectly provided access to this data.

The final issue (potentially) applies to all three problems: use of libraries that could lead to memory corruption. In this case, team SL-81 chose to use strcpy when processing user input, and in one instance failed to validate it, allowing an overflow. Rather than code this as Mistake, we considered it a bad choice because a safe function (strlcpy) could have been used instead to avoid the security issue.

4.2.2 Conceptual Error

Teams that chose a secure design often introduced a vulnerability in their implementation due to a conceptual misunderstanding (rather than a simple mistake). This Conceptual Error sub-type manifested in six ways.

Most commonly, teams used a fixed value when a random or unpredictable one was necessary (P=26, V=26). This included using hardcoded account passwords (P=8, V=8), encryption keys (P=3, V=3), salts (P=3, V=3), or using a fixed IV (V=12, N=12).

Listing 1: SC-76 Used a hardcoded IV seed.

```go
1 var nextNonce uint64 = 1337
2 ...
3 func sendMessage(conn *net.Conn, message []byte) (err error) {
4     var box []byte
5     var nonce [24]byte
6     byteOrder.PutUint64(nonce[:], nextNonce)
7     box = secretbox.Seal(box, message, &nonce, &sharedSecret)
8     var packet = Packet{Size: uint64(len(box)),
9         Nonce: nextNonce}
10     nextNonce++
11     writer := *conn
12     err = binary.Write(writer, byteOrder, packet)
13     ...}
```

Sometimes chosen values were not fixed, but not sufficiently unpredictable (P=7, V=8). This included using a timestamp-based nonce, but making the accepted window too large (P=3, V=3); using repeated nonces or IVs (P=3, V=4); or using predictable IVs (P=1, V=1). As an example, SC-76 attempted to use a counter-based IV to ensure IV uniqueness. Listing 1 shows that nonce nextNonce is incremented after each message. Unfortunately, the counter is re-initialized every time the client makes a new transaction, so all messages to the server are encrypted with the same IV. Further, both the client and server initialize their counter with the same number (1337 in Line 1 of Listing 1), so the messages to and from the server for the first transaction share an IV. If team SC-76 had maintained the counter across executions of the client (i.e., by persisting it to a file) and used a different seed for the client and server, both problems would be avoided.

Other teams set up a security mechanism correctly, but only protected a subset of necessary components (P=9, V=10). For example, Team SL-66 generated a MAC for each log entry separately, preventing an attacker from modifying an entry, but allowing them to arbitrarily delete, duplicate, or reorder log entries. Team SC-24 used an HTTP library to handle client-server communication, then performed encryption on each packet’s data segment. As such, an attacker can read or manipulate the HTTP headers; e.g., by changing the HTTP return status the attacker could cause the receiver to drop a legitimate packet.

In three cases, the team passed data to a library that failed to handle it properly (P=3, V=3). For example, MD-27 used an access-control library that takes rules as input and returns whether there exists a chain of delegations leading to the content owner. However, the library cannot detect loops in the delegation chain. If a loop in the rules exists, the library enters an infinite loop and the server becomes completely unresponsive. (We chose to categorize this as a Conceptual Error.)
5 Analysis of Vulnerabilities

This section considers the prevalence (RQ1) of each vulnerability type as reported in Table 2 along with the attacker control (RQ2), and exploitability (RQ3) of introduced types. Overall, we found that simple implementation mistakes (Mistake) were far less prevalent than vulnerabilities related to more fundamental lack of security knowledge (No Implementation, Misunderstanding). Mistakes were almost always exploited by at least one other team during the Break It phase, but higher-level errors were exploited less often. Teams that were careful to minimize the footprint of security-critical code were less likely to introduce mistakes.

5.1 Prevalence

To understand the observed frequencies of different types and sub-types, we performed planned pairwise comparisons among them. In particular, we use a Chi-squared test—appropriate for categorical data [34]—to compare the number of projects containing vulnerabilities of one type against the projects with another, assessing the effect size (φ) and significance (p-value) of the difference. We similarly compare sub-types of the same type. Because we are doing multiple comparisons, we adjust the results using a Benjamini-Hochberg (BH) correction [13]. We calculate the effect size as the measure of association of the two variables tested (φ) [24, 282-283]. As a rule of thumb, φ ≥ 0.1 represents a small effect, φ ≥ 0.3 a medium effect, and φ ≥ 0.5 a large effect [23]. A p-value less than 0.05 after correction is considered significant.

Teams often did not understand security concepts. We found that both types of vulnerabilities relating to a lack of security knowledge—No Implementation (φ = 0.29, p < 0.001) and Misunderstanding (φ = 0.35, p < 0.001)—were significantly more likely (roughly medium effect size) to be introduced than vulnerabilities caused by programming Mistakes. We observed no significant difference between No Implementation and Misunderstanding (φ = 0.05, p = 0.46). These results indicate that efforts to address conceptual gaps should be prioritized. Focusing on these issues of understanding, we make the following observations.

Unintuitive security requirements are commonly skipped. Of the No Implementation vulnerabilities, we found that the Unintuitive sub-type was much more common than its All Intuitive (φ = 0.44, p < 0.001) or Some Intuitive (φ = 0.37, p < 0.001) counterparts. The two more intuitive sub-types did not significantly differ (φ = 0.08, p = 0.32) This indicates that developers do attempt to provide security — at least when incentivized to do so — but struggle to consider all the unintuitive ways an adversary could attack a system. Therefore, they regularly leave out some necessary controls.

Teams often used the right security primitives, but did
not know how to use them correctly. Among the Misunderstanding vulnerabilities, we found that the Conceptual Error sub-type was significantly more likely to occur than Bad Choice ($\phi = 0.23$, $p = .003$). This indicates that if developers know what security controls to implement, they are often able to identify (or are guided to) the correct primitives to use. However, they do not always conform to the assumptions of “normal use” made by the library developers.

Complexity breeds Mistakes. We found that complexity within both the problem itself and also the approach taken by the team has a significant effect on the number of Mistakes introduced. This trend was uncovered by a poisson regression (appropriate for count data) [17, 67-106] we performed for issues in the Mistakes type.4

Table 3 shows that Mistakes were most common in the MD problem and least common in the SL problem. This is shown in the second row of the table. The log estimate (E) of 6.68 indicates that teams were 6.68 × more likely to introduce Mistakes in MD than in the baseline secure communication case. In the fourth column, the 95% confidence interval (CI) provides a high-likelihood range for this estimate between 2.90 × and 15.37 ×. Finally, the p-value of < 0.001 indicates that this result is significant. This effect likely reflects the fact that the MD problem was the most complex, requiring teams to write a command parser, handle network communication, and implement nine different access control checks.

Similar logic demonstrates that teams were only 0.06 × as likely to make a mistake in the SL problem compared to the SC baseline. The SL problem was on the other side of the complexity spectrum, only requiring the team to parse command-line input and read and write securely from disk.

Similarly, not implementing the secure components multiple times (Minimal Trusted Code) was associated with an 0.36 × decrease in Mistakes, suggesting that violating the “Economy of Mechanism” principle [68] by adding unnecessary complexity leads to Mistakes. As an example of this effect, MD-74 reimplemented their access control checks four times throughout the project. Unfortunately, when they realized the implementation was incorrect in one place, they did not update the other three.

Mistakes are more common in popular languages. Teams that used more popular languages are expected to have a 1.09 × increase in Mistakes for every one unit increase in popularity over the mean Popularity5 ($p = 0.009$). This means, for example, a language 5 points more popular than average would be associated with a 1.54 × increase in Mistakes. One possible explanation is that this variable proxies for experience, as many participants who used less popular languages also knew more languages and were more experienced.

Finally, while the LoC were found to have a significant effect on the number of Mistakes introduced, the estimate is so close to one as to be almost negligible.

### 5.2 Exploit Difficulty and Attacker control

To answer RQ2 and RQ3, we consider how the different vulnerability types differ from each other in difficulty to exploit, as well as in the degree of attacker control they allow. We distinguish three metrics of difficulty: our qualitative assessment of the difficulty of finding the vulnerability (Discovery Difficulty); our qualitative assessment of the difficulty of exploiting the vulnerability (Exploit Difficulty); and whether a competitor team actually found and exploited the vulnerability (Actual Exploitation). Figure 1 shows the number of vulnerabilities for each type with each bar divided by Exploit Difficulty, bars grouped by Discovery Difficulty, and the left and right charts showing partial and full attacker control vulnerabilities, respectively.

To compare these metrics across different vulnerability types and sub-types, we primarily use the same set of planned pairwise Chi-squared tests described in Section 5.1. When necessary, we substitute Fisher’s Exact Test (FET), which is more appropriate when some of the values being compared are less than five [33]. For convenience of analysis, we binned Discovery Difficulty into Easy (execution) and Hard (source, deep insight). We similarly binned Exploit Difficulty into Easy (single-step, few steps) and Hard (many steps, deterministic or probabilistic).

Misunderstandings are rated as hard to find. Identifying Misunderstanding vulnerabilities often required the attacker to determine the developer’s exact approach and have a good understanding of the algorithms, data structures, or libraries

---

4We selected initial covariates for the regression related to the language used, best practices followed (e.g., Minimal Trusted Code), team characteristics (e.g., years of developer experience), and the contest problem. From all possible initial factor combinations, we chose the model with minimum Bayesian Information Criteria—a standard metric for model fit [63]. We include further details of the initial covariates and the selection process in Appendix C, along with discussion of other regressions we tried but do not include for lack of space.

5The mean Popularity score was 91.5. Therefore, C—whose Popularity score of 92 was nearest to the mean—can be considered representative the language of average popularity.
they used. As such, we rated Misunderstanding vulnerabilities as hard to find significantly more often than both No Implementation (∅ = 0.52, p < 0.001) and Mistake (∅ = 0.30, p = 0.02) vulnerabilities.

Interestingly, we did not observe a significant difference in actual exploitation between the Misunderstanding and No Implementation types. This suggests that even though Misunderstanding vulnerabilities were rated as more difficult to find, sufficient code review can help close this gap in practice.

That being said, Misunderstandings were the least common type to be actually exploited by Break It teams. Specifically, using a weak algorithm (Not Exploited=3, Exploited=2), using a fixed value (Not Exploited=14, Exploited=12), and using a homemade algorithm (Not Exploited=1, Exploited=1) were actually exploited in at most half of all identified cases. These vulnerabilities presented a mix of challenges, with some rated as difficult to find and others difficult to exploit. In the homemade encryption case (SL-61), the vulnerability took some time to find, because the implementation code was difficult to read. However, once an attacker realizes that the team has essentially reimplemented the Wired Equivalent Protocol (WEP), a simple check of Wikipedia reveals the exploit. Conversely, seeing that a non-random IV was used for encryption is easy, but successful exploitation of this flaw can require significant time and effort.

**No Implementations are rated as easy to find.** Unsurprisingly, a majority of No Implementation vulnerabilities were rated as easy to find (V=42, 58% of No Implementations). For example, in the SC problem, an auditor could simply check whether encryption, an integrity check, and a nonce were used. If not, then the project can be exploited. None of the All Intuitive or Some Intuitive vulnerabilities were rated as difficult to exploit; however, 45% of Unintuitive vulnerabilities were (V=22). The difference between Unintuitive and Some Intuitive is significant (∅ = 0.38, p = 0.003), but (likely due to sample size) the difference between Unintuitive and All Intuitive is not (∅ = 0.17, p = 0.17).

As an example, SL-7 did not use a MAC to detect modifications to their encrypted files. This mistake is very simple to identify, but it was not exploited by any of the BIBIFI teams. The likely reason for this was that SL-7 stored the log data in a JSON blob before encrypting. Therefore, any modifications made to the encrypted text must maintain the JSON structure after decryption, or the exploit will fail. The attack could require a large number of tests to find a suitable modification.

**Mistakes are rated as easy to find and exploit.** We rated all Mistakes as easy to exploit. This is significantly different from both No Implementation (∅ = 0.43, p = 0.001) and Misunderstanding (∅ = 0.51, p < 0.001) vulnerabilities, which were rated as easy to exploit less frequently. Similarly, Mistakes were actually exploited during the Break It phase significantly more often than either Misunderstanding (∅ = 0.35, p = 0.001) or No Implementation (∅ = 0.28, p = 0.006). In fact, only one Mistake (0.03%) was not actually exploited by any Break It team. These results suggest that although Mistakes were least common, any that do find their way into production code are likely to be found and exploited. Fortunately, our results also suggest that code review may be sufficient to find many of these vulnerabilities. (We note that this assumes that the source is available, which may not be the case when a developer relies on third-party software.)

**No significant difference in attacker control.** We find no significant differences between types or sub-types in the incidence of full and partial attacker control. This result is likely partially due to the fact that partial attacker control vulnerabilities still have practically important consequences. Because of this fact, our BIBIFI did not distinguish between attacker control levels when awarding points; i.e., partial attacker control vulnerabilities received as many points as full attacker control. The effect of more nuanced scoring could be investigated in future work. We do observe a trend that Misunderstanding vulnerabilities exhibited full attacker control more often (V=50, 70% of Misunderstandings) than No Implementation and Mistake (V=44, 61% and V=20, 51%, respectively); this trend specifically could be further investigated in future stud-
ies focusing on attacker control.

6 Discussion and Recommendations

Our results are consistent with real-world observations, add weight to existing recommendations, and suggest prioritizations of possible solutions.

Our vulnerabilities compared to real-world vulnerabilities. While we compiled our list of vulnerabilities by exploring BIBIFI projects, we find that our list closely resembles both Mitre’s CWE and OWASP’s Top Ten [55, 61] lists. Overlapping vulnerabilities include: broken authentication (e.g., insufficient randomness), broken access control, security misconfiguration (e.g., using an algorithm incorrectly or with the wrong default values), and sensitive data exposure (e.g., side-channel leak).

API design. Our results support the basic idea that security controls are best applied transparently, e.g., using simple APIs [36]. However, while many teams used APIs that provide security (e.g., encryption) transparently, they were still frequently misused (e.g., failing to initialize using a unique IV or failing to employ stream-based operation to avoid replay attacks). It may be beneficial to organize solutions around general use cases, so that developers only need to know the use case and not the security requirements.

API documentation. API usage problems could be a matter of documentation, as suggested by prior work [4, 57]. For example, teams SC-18 and SC-19 used TLS socket libraries but did not enable client-side authentication, as needed by the problem. This failure appears to have occurred because client-side authentication is disabled by default, but this fact is not mentioned in the documentation.6 Defaults within an API should be safe and without ambiguity [36]. As another example, SL-22 (Listing 2) disabled the automatic integrity checks of the SQLCipher library. Their commit message stated “improve performance by disabling per-page MAC protection.” We know that this change was made to improve performance, but it is possible they assumed they were only disabling the “per-page” integrity check while a full database check remained. The documentation is unclear about this.7

Security education. Even the best documented APIs are useless when teams fail to apply security at all, as we observed frequently. A lack of education is an easy scapegoat, but we note that many of the teams in our data had completed a cybersecurity MOOC prior to the competition. We reviewed lecture slides and found that all needed security controls for the BIBIFI problems were discussed. While only three teams failed to include All Intuitive requirements (5% of MOOC teams), a majority of teams failed to include Unintuitive requirements (P=33, 55% of MOOC teams). It could be that the topics were not driven home in a sufficiently meaningful manner. An environment like BIBIFI, where developers practice implementing security concepts and receive feedback regarding mistakes, could help. Future work should consider how well competitors from one contest do in follow-on contests.

Vulnerability analysis tools. There is significant interest in automating security vulnerability discovery (or preventing vulnerability introduction) through the use of code analysis tools. Such tools may have found some of the vulnerabilities we examined in our study. For example, static analyses like SpotBugs/Findbugs [8, 41], Infer [16], and FlowDroid [9]; symbolic executors like KLEE [15] and angr [71]; fuzz tests like AFL [81] or libfuzzer [70]; and dynamic analyses like libdft [43] and TaintDroid [29] could have uncovered vulnerabilities relating to memory corruption, improper parameter use (like a fixed IV [25]), and missing error checks. However, they would not have applied to the majority of vulnerabilities we saw, which are often design-level, conceptual issues. An interesting question is how automation could be used to address security requirements at design time.

Determining security expertise. Our results indicate that the reason teams most often did not implement security was due to a lack of knowledge. However, neither years of development experience nor whether security training had been completed had a significant effect on whether any of the vulnerability types were introduced. This finding is consistent with prior research [60] and suggests the need for a new measure of security experience. Previous work by Votipka et al. contrasting vulnerability discovery experts (hackers) and nonexperts (software testers) suggested the main factor behind their difference in experience was the variety of different vulnerabilities they discovered or observed (e.g., read about or had described to them) [79]. Therefore, a metric for vulnerability experience based on the types of vulnerabilities observed previously may have been a better predictor for the types of vulnerabilities teams introduced.

7 Related Work

The original BIBIFI paper [66] explored how different quantitative factors influenced the performance and security of contest submissions. This paper complements that analysis with in-depth, qualitative examination of the introduced vulnerabilities in a substantial sample of BIBIFI submissions (including a new programming problem, multiuser database).

The BIBIFI contest affords analysis of many attempts at the same problem in a context with far more ecological validity than a controlled lab study. This nicely complements prior work examining patterns in the introduction and identification of vulnerabilities in many contexts. We review and compare to some of this prior work here.

---

7https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC
Measuring metadata in production code. Several researchers have used metadata from revision-control systems to examine vulnerability introduction. In two papers, Meneely et al. investigated metadata from PHP and the Apache HTTP server [50, 52]. They found that vulnerabilities are associated with higher-than-average code churn, committing authors who are new to the codebase, and editing others’ code rather than one’s own. Follow-up work investigating Chromium found that source code reviewed by more developers was more likely to contain a vulnerability, unless reviewed by someone who had participated in a prior vulnerability-fixing review [51]. Significantly earlier, Sliverski et al. explored mechanisms for identifying bug-fix commits in the Eclipse CVS archives, finding, e.g., that fix-inducing changes typically span more files than other commits [73]. Perl et al. used metadata from Github and CVEs to train a classifier to identify commits that might contain vulnerabilities [62].

Other researchers have investigated trends in CVEs and the National Vulnerability Database (NVD). Christey et al. examining CVEs from 2001–2006, found noticeable differences in the types of vulnerabilities reported for open- and closed-source operating-system advisories [22]. As a continuation, Chang et al. explored CVEs and the NVD from 2007–2010, showing that the percentage of high-attacker control vulnerabilities decreased over time, but that more than 80% of all examined vulnerabilities were exploitable via network access without authentication [21]. We complement this work by examining a smaller set of vulnerabilities in more depth. While these works focus on metadata about code commits and vulnerability reports, we instead examine the code itself.

Measuring cryptography problems in production code. Lazar et al. discovered that only 17% of cryptography vulnerabilities in the CVE database were caused by bugs in cryptographic libraries, while 83% were caused by developer misuse of the libraries [46]. This accords with our Conceptual Error results. Egele et al. developed an analyzer to recognize specific cryptographic errors and found that nearly 88% of Google Play applications using cryptographic APIs make at least one of these mistakes [28]. Kruger et al. performed a similar analysis of Android apps and found 95% made at least one misuse of a cryptographic API [45]. Other researchers used fuzzing and static analysis to identify problems with SSL/TLS implementations in libraries and in Android apps [30, 35]. Focusing on one particular application of cryptography, Reaves et al. uncovered serious vulnerabilities in mobile banking applications related to homemade cryptography, certificate validation, and information leakage [64]. These works examine specific types of vulnerabilities across many real-world programs; our contest data allows us to similarly investigate patterns of errors made when addressing similar tasks, but explore more types of vulnerabilities.

Controlled experiments with developers. In contrast to production-code measurements, other researchers have explored security phenomena through controlled experiments with small, security-focused programming tasks. Oliveira et al. studied developer misuse of cryptographic APIs via Java “puzzles” involving APIs with known misuse cases and found that neither cognitive function nor expertise correlated with ability to avoid security problems [60]. Other researchers have found, in the contexts of cryptography and secure password storage, that while simple APIs do provide security benefits, simplicity is not enough to solve the problems of poor documentation, missing examples, missing features, and insufficient abstractions [4, 56–58]. Perhaps closest to our work, Finifter et al. compared different teams’ attempts to build a secure web application using different tools and frameworks [31]. They found no relationship between programming language and application security, but that automated security mechanisms were effective in preventing vulnerabilities.

Other studies have experimentally investigated how effective developers are at looking for vulnerabilities. Edmundson et al. conducted an experiment in manual code review: no participant found all three previously confirmed vulnerabilities, and more experience was not necessarily correlated with more accuracy in code review [27]. Other work suggested that users found more vulnerabilities faster with static analysis than with black-box penetration testing [69].

We further substantiate many of these findings in a different experimental context: larger programming tasks in which functionality and performance were prioritized along with security, allowing increased ecological validity while still maintaining some quasi-experimental controls.

8 Conclusion

Secure software development is challenging, with many proposed remediations and improvements. To know which interventions are likely to have the most impact requires understanding which security errors programmers tend to make, and why. To this end, we presented a systematic, qualitative study of 94 program submissions to a secure-programming contest, each implementing one of three non-trivial, security-relevant programming problems. Over about six months, we labeled 182 unique security vulnerabilities (some from the 866 exploits produced by competitors, some we found ourselves) according to type, attacker control, and exploitability, using iterative open coding. We also coded project features aligned with security implementation. We found implementation mistakes were comparatively less common than failures in security understanding—78% of projects failed to implement a key part of a defense, or did so incorrectly, while 21% made simple mistakes. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.
References


[34] Karl Pearson F.R.S. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. Philosophical Magazine, 50(302):157–175, 1900.


A Additional Contest Details

To provide additional context for our results, this appendix includes a more thorough breakdown of the sampled population along with the number of breaks and vulnerabilities for each competition. Specifically, Table 4 presents statistics for several team, participant demographic characteristics, and counts of break submissions and unique vulnerabilities introduced divided by competition and Figure 2 shows the variation in team sizes across the four competitions.
### Table 4: Demographics of participants from sampled teams along with the number of breaks submitted and vulnerabilities introduced per competition. Some participants declined to specify gender. Slashed values represent mean/min/max.

<table>
<thead>
<tr>
<th>Contest</th>
<th>Fall 14 (SL)</th>
<th>Spring 15 (SL)</th>
<th>Fall 15 (SC)</th>
<th>Fall 16 (MD)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># Teams</td>
<td>10</td>
<td>42</td>
<td>27</td>
<td>15</td>
<td>94</td>
</tr>
<tr>
<td># Contestants</td>
<td>26</td>
<td>100</td>
<td>86</td>
<td>35</td>
<td>247</td>
</tr>
<tr>
<td>% Male</td>
<td>46 %</td>
<td>92 %</td>
<td>87 %</td>
<td>80 %</td>
<td>84 %</td>
</tr>
<tr>
<td>% Female</td>
<td>12 %</td>
<td>4 %</td>
<td>8 %</td>
<td>3 %</td>
<td>6 %</td>
</tr>
<tr>
<td>Age</td>
<td>22.9/18/30</td>
<td>35.3/20/58</td>
<td>32.9/17/56</td>
<td>24.5/18/40</td>
<td>30.1/17/58</td>
</tr>
<tr>
<td>% with CS degrees</td>
<td>85 %</td>
<td>39 %</td>
<td>35 %</td>
<td>57 %</td>
<td>45 %</td>
</tr>
<tr>
<td>Years programming</td>
<td>2.9/1/4</td>
<td>9.7/0/30</td>
<td>9.6/2/37</td>
<td>9.6/3/21</td>
<td>8.9/0/37</td>
</tr>
<tr>
<td>Team size</td>
<td>2.6/1/6</td>
<td>2.4/1/5</td>
<td>3.2/1/5</td>
<td>2.3/1/8</td>
<td>2.7/1/8</td>
</tr>
<tr>
<td># PLs known per team</td>
<td>6.4/3/14</td>
<td>6.9/1/22</td>
<td>8.0/2/17</td>
<td>7.9/1/17</td>
<td>7.4/1/22</td>
</tr>
<tr>
<td>% MOOC</td>
<td>0%</td>
<td>100 %</td>
<td>91 %</td>
<td>53 %</td>
<td>76 %</td>
</tr>
<tr>
<td># Breaks</td>
<td>30</td>
<td>334</td>
<td>242</td>
<td>260</td>
<td>866</td>
</tr>
<tr>
<td># Vulnerabilities</td>
<td>12</td>
<td>41</td>
<td>64</td>
<td>65</td>
<td>182</td>
</tr>
</tbody>
</table>

Figure 2: Histogram of team size split by competition.

### B Additional Coding

In addition to the project variables identified to have significant effect on the types of vulnerabilities introduced, we coded several other variables. This appendix describes the full set of project variables we coded. Table 5 provides a summary of all the variables.

One potential reason for vulnerabilities introduction is due to hard to understand code. If team members cannot comprehend the code, then misunderstandings could occur that lead to more vulnerabilities. To determine whether this was the case, we coded each project according to several readability measures. These included whether the project was broken into several single-function sub-components (Modularity), whether the team used variable and function names representative of the semantic role of the associated variable or function (Variable Naming), whether whitespace was used to make it easy to visualize control-flow and variable scope (Whitespace), and whether comments were included throughout the program to summarize relevant details (Comments).

In addition to the readability metrics, we identified whether each project followed common secure development best practices [14, pg. 32-36], specifically Economy of Mechanism and Minimal Trusted Code.

When coding Economy of Mechanism, if the reviewer judged that the project only included necessary steps to provided the intended security properties, then the project’s security was marked as economical. For example, one project submitted to the secure log problem added a constant string to the end of each access log event before encrypting. In addition to using a message authentication code to ensure integrity, they checked that this hardcoded string was unchanged after decryption as part of their integrity check. System vulnerable if a known-plaintext attack exists. Because removing this unnecessary additional step would not sacrifice security, we coded this project was not marked as economical.

**Minimal Trusted Code** was measured by checking whether the security-relevant functionality was implemented in multiple locations in the codebase. Projects passed this check if they created a single function for each security function (e.g., encryption, access control checks, etc.) and called that function throughout the project. The alternative—copying and pasting code wherever security functionality was needed—is likely to lead to mistakes if each code segment is not updated whenever changes are necessary.

### C Regression Analysis

For each subclass of vulnerability types, we performed a poisson regression [17, 67-106] to understand whether any of the team’s characteristics or their programming decisions influenced the vulnerabilities they introduced. In this appendix, we provide an extended discussion of this analysis, focusing on the full set of covariates included in each initial model, our model selection process, and the results omitted from...
<table>
<thead>
<tr>
<th>Variable</th>
<th>Levels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular</td>
<td>T / F</td>
<td>Whether the project is segmented into a set of functions and classes each performing small subcomponents of the project</td>
</tr>
<tr>
<td>Variable Naming</td>
<td>T / F</td>
<td>Whether the author used variable names that indicated the purpose of the variable</td>
</tr>
<tr>
<td>Whitespace</td>
<td>T / F</td>
<td>Whether the author used whitespace (i.e., indentation and new lines) to allow the reader to easily infer control-flow and variable scope</td>
</tr>
<tr>
<td>Comments</td>
<td>T / F</td>
<td>Whether the author included comments to explain blocks of the project</td>
</tr>
<tr>
<td>Economy of Mechanism</td>
<td>T / F</td>
<td>How complicated are the implementations of security relevant functions</td>
</tr>
<tr>
<td>Minimal Trusted Code</td>
<td>T / F</td>
<td>Whether security relevant functions are implemented once or re-implemented each time they are needed</td>
</tr>
</tbody>
</table>

Table 5: Summary of the project codebook.

the main paper due to their lack of significant results or poor model fit.

C.1 Initial Covariates

As a baseline, all regression models included factors related to the language used (Type Safety and Popularity), team characteristics (development experience and security education), and the associated problem in each initial model. These base covariates were used to understand the effect of a team’s intrinsic characteristics, their development environment, and the specific problem specification. The Type Safety variable identified whether each project was statically typed (e.g., Java or Go, but not C or C++), dynamically typed (e.g., Python, Ruby), or C/C++ (Type Safety).

For Misunderstanding regressions—one for each subclass—the Bad Choice regression only included the baseline covariates and the Conceptual Error regression added the type of the library (Library Type) to see if the type of library used had an effect on developer success. The project’s Library Type was one of three categories based on the libraries used (Library Type): no library used (None), a standard library provided with the language (e.g., PyCrypto for Python) (Language), or a non-standard library (3rd Party).

The No Implementation regressions only included the base set of covariates. Additionally, because the Some Intuitive vulnerabilities only occurred in the MD problem, we did not include problem as a covariate in the Some Intuitive regression.

In addition to these baseline covariates, the Mistake regression added the Minimal Trusted Code and Economy of Mechanism variables, whether the team used test cases during the build phase, and the number of lines of code in the project. These additional covariates were chosen as we expect smaller, simpler, and more rigorously tested code to include less mistakes.

C.2 Model Selection

We calculated the Bayesin Information Criterion (BIC)—a standard metric for model fit [63]—for all possible combinations of the initial factors. To determine the optimal model and avoid overfitting, we selected the minimum BIC model. Because our study is only semi-controlled, there are a large number of covariates which must be accounted for in each regression. Therefore, our regressions were only able to identify large effects [23]. Note, for this reason, we also did not include any interaction variables in our regressions. Including interaction variables would have reduced the power of each model significantly and precluded finding even very large effects. Further, due to the sparse nature of our data (e.g., many languages and libraries were used, in many cases only by one team), some covariates could only be included in an aggregated form, which limits the specificity of the analysis. Future, more focused work should consider these interactions and more detailed questions.

C.3 Results

Tables 6–10 provide the results of each of the regressions that were not included in the main text of the paper.

D Revisions from Prior Submissions

We thank the reviewers for their thoughtful comments. We have made several changes to the paper based on this feedback. In this appendix, we begin by addressing the issue of biased sampling raised as the main reason for the major revision.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Log Value</th>
<th>Estimate</th>
<th>CI</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOOC</td>
<td>False</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>1.76</td>
<td>[0.70, 4.34]</td>
<td>0.23</td>
<td></td>
</tr>
</tbody>
</table>

*Significant effect

– Base case (Estimate=1, by definition)

Table 7: Summary of regression over Conceptual Error vulnerabilities. Pseudo $R^2$ measures for this model were 0.01 (McFadden) and 0.02 (Nagelkerke).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Log Value</th>
<th>Estimate</th>
<th>CI</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yrs. Experience</td>
<td>8.9</td>
<td>1.12</td>
<td>[0.82, 1.55]</td>
<td>0.47</td>
<td></td>
</tr>
</tbody>
</table>

*Significant effect

– Base case (Estimate=1, by definition)

Table 8: Summary of regression over All Intuitive vulnerabilities. Pseudo $R^2$ measures for this model were 0.06 (McFadden) and 0.06 (Nagelkerke).

decision. Then we address each reviewer’s comments point by point. We conclude with a list of additional changes made to improve readability and to discuss recent work that has been published since initial review.

D.1 Sampling Procedure

The reviewers (notably reviewer D) raised the concern that projects with no submitted breaks might be meaningfully different than those with submitted breaks, such that sampling them unevenly (more projects with breaks than without) could introduce bias. Moreover, there was concern that while sampling according to vulnerability frequency might be useful, we sampled according to break submission frequency, which is not necessarily the same thing. (This concern is repeated in full in Appendix D.5, which responds to reviewer D.)

To remedy this, we randomly selected and analyzed 18 additional projects that did not have any breaks submitted against them. As a result, we have now analyzed 67% of projects with submitted breaks and 67% of those without. This brought our total of projects analyzed to 94. Because our initial set of projects and this new set were both randomly sampled, this additional sampling does not affect the randomness of the total procedure. That is, there is no meaningful difference between our sampling which occurred at two distinct times (before and after the initial review) and a single sampling of 67% of all projects in each category.

Within the 18 added projects, we identified 10 additional vulnerabilities that were not submitted as breaks by BIBIFI participants, bringing our total number of vulnerabilities to 182. Then we performed a post-hoc spearman rank correlation test [80, pg. 508] to examine whether the number of breaks identified by participants (the metric we used for sampling) was correlated with the underlying number of identified vulnerabilities. We found that these two variables were highly correlated, indicating that sampling based on the number of breaks is a reasonable method for ensuring coverage of vulnerability types. Most importantly, this additional analysis did not result in any changes in our findings. We observed minor changes in the raw numbers, but no differences in the overall trends. The vulnerabilities we identified in the added projects matched our expectations based on the previously reported results, increasing our confidence in the validity of our results.

The substance of the above information is presented in Section 3.2.1.

The analysis of these 18 additional projects precipitated several other modifications to the paper:

- We updated the Abstract and Sections 1 and 8 to indicate the percentage of projects that introduced vulnerabilities due to simple mistakes (21%) and misunderstandings of security concepts (78%).
- We updated the Program Characteristics paragraph of Section 2.2 to include statistics for additional projects.
- We updated Table 2 to reflect new counts and percentages of observed vulnerabilities.
- We updated numbers shown throughout Section 4 to reflect new counts of observed vulnerabilities.
- We updated Figure 1 to reflect the new attacker control and difficulty to find and exploit counts.
- We updated Section 5 to reflect small changes to the calculated statistics for vulnerability trends.
- We updated Appendix C to reflect minor changes in the regression results (parameter values changed slightly but no overall findings changed).
D.2 Reviewer A

• **Intro - MOOC - not everyone knows what that is.**
  We added the spelled out version of the acronym in Section 1

• **2.3 - Great section. I'm assuming where applicable this applies to the set of projects you selected (sometimes you call that out explicitly but, with team size, you do not).**
  That is correct. For brevity, we only specifically call out problems when our statements do not apply to all of them.

• **Teams free to choose programming languages - limits ecological validity to that sort of independence (not found in most organizations doing programming). You call out something similar in your penultimate bullet in that section.**
  We agree that this decision is not always made by the end-line developer, but is instead sometimes dictated by their company. We updated the sixth bullet of Section 2.3 to include this limitation. While we consider issues associated with companies prescribing specific languages interesting for future work, we believe there is value in creating a baseline by first investigating developer decisions when they are allowed free choice (e.g., our work might justify companies’ making decisions for the developers)

• **Median of 12 years experience looks solid. A more detailed breakdown would be helpful. It would help this area to build up demographic expectations and goals on this dimension.**
  We added Table 4 in Appendix A with an more in-depth breakdown of team and participant characteristics. Additionally, while calculating these statistics, we realized that our original calculation of mean years of experience was incorrect. Our mean years of development experience is actually 8.9 years.

• **I was looking for a full breakdown in team sizes. Giving the range and median makes clear these are small independent teams. Is there anything interesting in the full breakdown of team sizes for future researchers?**
  We added Figure 2 in Appendix A, which shows counts for the number of teams of each team size split by competition.

• **Penultimate bullet - there are a number of other issues where sw eng in large organizations is different these days. Which would flesh out the discussion in 2.3. One I can think of offhand is that in SaaS sw developers are responsible for understanding and working with the full deployment tool suite and lifecycle. "Developer centered security and the symmetry of ignorance" touches on that, and might have other areas of note.**
  We agree that BIBIFI is not representative of all aspects of the complex environment developers are asked to operate in. We have updated the sixth bullet of Section 2.3 to state that we focus on the task of creating a program from scratch and do not consider other aspects of the operational lifecycle. We believe our setting is an important baseline for a broader exploration in future work.

• **Your last bullet seems related to your 3rd to last bullet. Companies that have security experts prime their employees in potentially similar ways.**
  We added a statement to the last bullet of Section 2.3 to point out this similarity.

• **3.1 - table references unbound**
  This table reference has been fixed and now points to Table 1.

• **3.1.2 - would be useful to future researchers to know what project features you coded that did not have an effect. Perhaps you would put them in an appendix.**
  We added Appendix B with a detailed description of all the items in the project codebook that were not found to have an effect.

• **I appreciate the detailed 3.2.1. I was still left wondering why each of the "stop" numbers (30% of breaks, 25% break less, 56 with one, and 20 with none). Time constraints?**
  We chose analyze a sample of projects rather than all of them due to the time-intensive nature of the work. We updated the discussion in Section 3.2.1 to explain that we chose to analyze a random sample of 67% of projects, bucketed based on the count of vulnerabilities identified in the contest, as a reasonable tradeoff between representativeness and time to complete the analysis.

• **4.2.2 - top of 2nd column, referencing your discussion in 2.3 of how large organizations have security support and specialists, crypto is often (perhaps always) a focus area of those processes and specialists, for just this reason (conceptually, it remains difficult and tricky). In large orgs, developers working with crypto will have required tools, patterns, reviews, and/or always be specialists.**
  We agree that this is one of the areas where our competition differs from larger organizations. Indeed, our study constitutes a baseline result from which one could attempt to justify the use of such security experts, or other interventions, e.g., API and documentation improvement. Both can improve security both for security mature organizations and less mature companies.

• **Section 5, last sentence - you say that this section is about Table 2, but I could not figure out what part of Table 2 had information on which teams were careful to minimize security-critical footprint. I think that is not covered there. It’s in Table 3?**
The wording of the first sentence of Section 5 was confusing. We have updated it so that is clear that Table 2 only includes information about the prevalence of each vulnerability class. Data regarding the attacker control allowed, exploitability, and other factors is not included in Table 2, but instead referenced as necessary throughout the section.

• 5.1. “Unintuitive security” - “at least when incentivized to do so”. I would argue that this context goes beyond incentivizing security. It is “part of the job”. That’s not even the same as a $4k shared team bonus; it’s your job and salary. There is some qualitative analysis in the SOUPS community of software developers and aspects such as this. See for example https://www.usenix.org/conference/soups2018/presentation/haney-mindsets

In Section 2.3, we added a sentence to the end of the fourth bullet point discussing the fact that a $4000 team prize may not be representative of security-mature companies, but that there are many companies which are not security mature.

• “Complexity breeds Mistakes” This insight goes back to “The Protection of Information in Computer Systems” and is called “Economy of Mechanism” there. I recommend referencing it, as it is useful to know when long standing principles are supported, and when they are disproved.

We have added a reference to this citation to Section 5.1.

• Last paragraph of that section, first sentence, “Similarly, implementing” - surely there is a “not” missing between those two words.

We have added a “not” to this sentence to fix this typo.

• “Misunderstandings are hard to find” 2nd paragraph - I would suggest an alternative hypothesis for future consideration. Often a security/vulnerability test is not nearly as complicated as understanding and fixing the problem. Developers on the other end of a vulnerability report see this a lot.

We agree that fixing a problem based on a vulnerability report can be much more difficult than finding the vulnerability in the first place. This may be one reason for low fixing rates shown in prior analysis of the BIBIFI data. However, this is out of scope for this paper, which only considers the vulnerabilities introduced in each project and not developers’ fixing behaviors. We leave further discussion to future work.

• On a related note, while I understand why a contest would require an exploit to identify a vulnerability, this would be a huge problem as a requirement in securely developing code. It explodes the resources required to use good practice to avoid previously discovered problems. Another place where this context is not representative of larger development organizations.

We agree that exploits are not a requirement of vulnerability reporting. Our focus in this paper is not on the way vulnerabilities are found or reported in the targeted software, but on the form and reasons for those vulnerabilities. As such, we can get a more complete picture by considering both exploits submitted during a contest and vulnerabilities identified by the researchers during analysis. We added a bullet in Section 2.3 to discuss this.

• ”Mistakes are easy...” - I think you’re assuming open source. Teams did have access to the source for the break it phase. Many products and services are not open source.

We added a sentence to Section 5.2 indicating the assumption that source is available.

• “No significant difference in severity” - this may be due to no incentives for severity. Which is unlike the real world in organizations (both CVSS and internal severities for internally found vulnerabilities).

We added further discussion to Section 5.2 indicating this limitation in the competition’s design and suggesting directions for future work.

• Section 5 - API documentation - last line - I can’t disagree, but that’s a battle that’s been fought and lost over decades. It’s surprisingly hard to high-light all security-critical assumptions. If you believe that’s a tractable recommendation, give some references on how to do it, or who does it well.

We agree that this is a difficult problem. Due to space limitations we chose not to add anything further.

• “Vulnerability Analysis Tools” - last sentence - needs one more sentence on why that might be a fruitful area.

We are not sure exactly what the reviewer is looking for. On reflection, this is a potentially huge topic for discussion, and a very speculative one at that. So we replaced the last sentence with a more general comment.

• Next section - missing a reference for Votipka et al.

We added the missing reference.

• Bibliography - why no URLs for the things that must be URLs (since they don’t seem to be anything else).

The bug in the LaTeX code has been fixed to display URLs for these references.

D.3 Reviewer B

• while students are commonly and justifiably used as proxies for professionals, insights gained by students with limited development and security experience limit the types of novel conclusions that can be drawn (i.e., results simply confirm prior findings)
We agree that our population sample has drawbacks, as discussed in 2.3. That said, our MOOC-heavy population diversifies the traditional undergraduate population, as evidenced (for example) by their mean 8.9 years of reported development experience. Further, prior work suggests that in at least some secure-development studies, only security experience, not general development experience, correlates with security outcomes [5, 6, 56, 58]. This means students can offer an effective substitute to professionals.

More importantly, as the fourth paragraph of Section 1 now highlights, BIBIFI’s structure is such that as an object of study it complements both field surveys and traditional lab studies. This is evident both in the novel findings produced — e.g., mistakes are less likely and easier to find than vulnerabilities caused by misunderstanding of security concepts and unintuitive requirements are more likely to be missed — and strengthening and expansion of prior results through their confirmation in the BIBIFI context, which offers a useful mix of experimental control and ecological validity.

- **How should we train developers to enhance their practical knowledge and application of secure coding practices and proper security mechanisms to limit the introduction of vulnerable code?** That’s a question that I think this type of study could answer. The recommendations made however seem to rehash common

Our results suggest that standard educational interventions — like the MOOC many participants completed (which had lectures and projects) — are insufficient on their own. An environment like BIBIFI, where developers practice implementing security concepts and receive feedback regarding mistakes, could help; we plan to research this in future work. In the Security Education paragraph of Section 6, we add a discussion of why BIBIFI itself may be a useful method for educating developers to produce more secure software.

- **I do see these types of studies as valuable given the extension effort applied to coding the vulnerabilities and explanations gathered that are difficult to acquire by other means.**

- **It would be interesting to see if developers / team that do well in the ‘break-it’ portion of the competition (i.e., those well-versed at identifying and exploiting vulnerable code) perform better at writing ‘secure’ code that the field. Also, did you consider the score of the teams in different phases (score in build vs score in break) while coding the vulnerabilities? It would also be interesting to examine and compare teams who scored high in build but low in break and vice versa. This could potentially give insights about developer mindset (white hat developer vs functional developer).**

These are interesting questions. However, the relationship between break success and vulnerability introduction is noisy; breaking is measured after building and participant knowledge can change as they are exposed to other teams’ code. For space, we focused on more concrete factors.

- **Do developers / teams for the Break-it portion write code that susceptible to the same vulnerabilities that they have identified in other code?**

This is an interesting question. Due to the noisiness of the relationship between building and breaking, we leave answering it to future work.

- **need answers that provide guidance about which interventions, tools, policy and education are most effective - as a contribution Our contribution is in identifying the kinds of vulnerabilities and errors that interventions should focus on (and how they occur). In Section 6, we speculate on which interventions to try, but further research is needed to evaluate these ideas.**

- **are the types of vulnerabilities covered specified in CVE/NVD / OWASP - do the most common student vulnera- abilities match the most common types of vulnerabilities found in the wild**

We did observe significant overlap with the types of vulnerabilities found in other reporting. We added a discussion to the start of Section 6 about this.

- **severity category used should be known (maybe CVSS and how your category compares to CVSS)**

First, we chose to rename severity to attacker control to more clearly indicate that we are measuring how much control the attacker is able to gain over the data or program. For example, if the attacker can make changes to only part of the network transmitted data, then they only have partial control.

We include attacker control and exploit difficulty as key variables in our coding scheme because they map to the relevant CVSS concepts of Impact, Attack Complexity, and Exploitability. We chose to analyze them separately to provide a more in-depth analysis rather than a summary score. We did not include further components of CVSS as they were not relevant to the specific context of the competition. Instead of the pre-existing CVSS levels for these factors, we followed qualitative best practices of letting natural (anti)patterns emerge from the data, modifying the categorizations we apply accordingly. In Section 3.1.1, we explain these choices and the relationship between our coding and the CVSS score.

- **Why not apply static analysis tools that identify vulner- abilities to the codebase? That is, to determine the types of vulnerabilities found to confirm or add to your manual coding process.**

As stated in Section 6, few of the errors we identified are even targeted by analysis tools. Indeed, the original BIBIFI
paper found that break-it teams that used tools gained little benefit, in terms of bugs found [66].

- Good to cover factors that are against representativeness of the participants and study constraints (i.e., limitations to generalizability)

- The authors presented their research questions in section 3 but did not present the findings for these questions in a systematic and coherent way. For example, they presented their coding results in section 4 (which I suppose was the answer for Q1 - but not stated explicitly like the other?) but did not present answers for Q2 & 3 not until section 5.2, after presenting their a statistical test.

To make the relationship between our research questions and results more clear, we added explicit references at the beginning of Sections 4 and 5.1 to the associated research questions.

- The authors did not provide clear explanation why they chose the particular type of statistical tests and models and what they wanted to accomplish with it. Especially table 3 was particularly confusing as it introduced new terms (popularity, LoC) without even introducing them properly. Similarly they also introduced the effect of complexity out of nowhere. Their footnotes were kind of abrupt and did not help reader to understand the context.

We added labels along with the definitions of popularity and LoC into Section 3.1.2 to reduce confusion. We also attempted to clarify the discussion of the statistical methods we used at the start of 5.1 and 5.2.

- Choice of projects: The project distribution across three types of implementation problems (Secure communication, secure log, database) was not uniform, why (I may have missed the explanation)? Besides considering "number of breaks" per submitted solution from each team, why not consider other criteria like e.g. team size, team composition (all students vs all professionals) etc.? Also, what was the distribution of 'breaks' in each category of implementation problems?

Participation varied across competitions, causing non-uniform numbers of solutions to each problem (see original BIBIFI paper [66] for specifics and Table 4 for a breakdown of per competition statistics). We sampled proportionally to ensure representativeness of each problem, and representativeness of break distribution, as described in Section 3.2.1. The focus of our study was the software itself, so we organized sampling around that (rather than sampling equally from differing team sizes, say). We did consider features of the software itself, and how it was developed, in our analysis, as did the original BIBIFI paper (which discusses the distribution of breaks in each category).

- I actually enjoyed reading this paper as good qualitative insights can be difficult to come by. I also think that there is more to tease out of the analysis than what’s offered. We are constrained by space limitations, but if the reviewer has specific suggestions we would be happy to consider including them.

- Section 3.1 - missing table numbers
  This table reference has been fixed and now points to Table 1.

D.4 Reviewer C

- Overall, I liked reading this paper. I felt the analysis was well done, and the results reinforce the existing body of literature showing that API designs & documentation are falling short of what developers need.

- Initially, I was skeptical that BIBIFI competitions would yield generalizable or interesting results. However, Section 2.3 convinced me (particularly the fact that most competitors are post-degree professionals). That section is clear, well-written, and compelling. The authors do a good job framing how this research can be interpreted.

- The main drawback of the paper is that neither the technique nor the conclusions are particularly novel. The authors cite related work appropriately, but nonetheless the findings are fairly incremental. There was already an initial paper on BIBIFI competitions. This paper takes a slightly different approach, but the overlap takes away from the novelty. The conclusions are also not especially novel, although I do think this paper could serve as a nice canonical citation for "simple/obvious mistakes aren’t that common, most mistakes are more complex."

The original BIBIFI paper did not systematically look at the software artifacts themselves; rather, it considered a few features of those artifacts, such as programming language used, or team size. The present paper’s contribution is different: It presents a systematic, manual analysis of the software artifacts, to understand the form and prevalence of the vulnerabilities present. This analysis reveals new insights—such as the quantitative difference between mistakes (infrequent) and higher-level flaws (common)—and adds new weight to confirm existing ones (e.g., that API design and poor documentation can lead to security mistakes).

That BIBIFI is a unique vantage point (per the response to reviewer B’s first point above), should add weight to the evidence it provides. We have tried to make this point in paragraph 4 of the Introduction, and afterward.

- The table references in Section 3.1 are broken.
  This table reference has been fixed and now points to Table 1.
Why didn’t you use any existing industry standard rating systems (such as for VRPs) for the vulnerability scoring system?

We include attacker control (previously referred to as severity) and exploit difficulty as key variables in our coding scheme because they map to the relevant CVSS concepts of Impact, Attack Complexity, and Exploitability. We chose to analyze them separately to provide a more in-depth analysis rather than a summary score. We did not include further components of CVSS as they were not relevant to the specific context of the competition. Instead of the pre-existing CVSS levels for these factors, we followed qualitative best practices of letting natural (anti)patterns emerge from the data, modifying the categorizations we apply accordingly. In Section 3.1.1, we explain these choices and the relationship between our coding and the CVSS score.

D.5 Reviewer D

Expanding on the second “weaknesses” bullet above, I note on page 5 that you discuss primarily sampling from projects with submitted breaks. This biases the results based on the findings that students in the course could make, and the students are not experts in security (they are the same group of people who submitted the systems for analysis). Given this, I am worried that the actual distribution of vulnerabilities / weaknesses in the full corpus of program submissions is different* than the distribution of vulnerabilities / weaknesses in the samples that you studied.

Let’s call the set X the set of programs with submitted vulnerabilities, and the set Y the set of programs for which there were no submitted vulnerabilities.

I do see that you took a weighted random sample of the systems with submitted vulnerabilities (set X), such that you got coverage of 30% of the submitted vulnerabilities. But, as noted above, the submitted vulnerabilities is a metric provided by students in the class, and not by experts, and hence the weighted sampling that you use is weighted *not* by the actual coded vulnerabilities that you report on later. If you did a weighted selection based off of coded vulnerabilities, after you independently coded for vulnerabilities, as experts, then this weighted random selection process might make more sense.

On top of that, I am worried that you then combined the 56 projects with submitted vulnerabilities (the subset X’ of X) with 20 projects from the no-submitted vulnerabilities set (the subset Y’ of Y) in such a way that your ultimate results would be influenced by the differences between the vulnerabilities that students submitted and the vulnerabilities that you found.

Now on to what I suggest doing about this. I suggest *not* bundling the 56 programs in set X’ with the 20 programs in set Y’. Or if you do, use things like staked bar graphs or otherwise keep the analysis of these sets separate. Also, I suggest analyzing to see if there is a difference between the distribution of vulnerabilities that students disclose with the distribution of vulnerabilities that you found, as experts.

We appreciate your concern about the sampling bias and took it quite seriously in our revision. We primarily discuss our response to this point in the main response to the revision requirements, above. To explain a bit further, it was not possible to sample on the vulnerabilities we coded, because until we coded each project we didn’t know how many vulnerabilities we would find—we would have had to analyze all projects, negating the time savings benefit of sampling. Instead, we sampled based on the submitted breaks, making an assumption that this would correlate with actual distribution of vulnerabilities. As you suggested, we have now confirmed that this assumption was reasonable, as the submitted breaks are strongly correlated with total vulnerabilities including ones we identified that competitors didn’t (Section 3.2.1). If we had instead found that these two metrics diverged, we would have thrown out our initial sampling hypothesis and continued to analyze the remaining projects.

To further resolve this issue, we sampled more projects from the set with no breaks submitted, bringing us to uniform sampling (67%) of projects in all categories. None of our findings changed as a result of the additional analysis. At this point, we are confident that our sample is representative and it is reasonable to consider all projects and vulnerabilities together in the main analysis.

Returning to my first weakness bullet, even if programmers are trying to be secure for a contest, the process of competing in a contest is different than in a real development environment. Further, what the competitors do and do not do from a security perspective might be more an indication of the instructor not programmers in general. Did you notice difference between different groups of students?

As we state in Section 2.3, while we agree that the contest setting is not a perfect proxy for real world development, we believe it provides a useful approximation while allowing the study of developer behaviors that would be impossible in a field study or laboratory setting.

These competitions were carried out over the course of four semesters with participants from several different educational backgrounds (only a portion of participants came from our MOOC). We did not observe any differences between participant groups.

Minor note, but on page 4, when you talk about “full compromise”, I would consider a full compromise to be root access, not message corruption.

We have updated Section 3.1.1 to be more clear on our specific definition of “full compromise” as it relates to the targeting of specific information or systems within the competition context.
D.6 Additional revisions

In addition to changes made based on the reviewers’ feedback, we made further revisions to improve readability, discuss related work and trim the text to meet space constraints. The following are the most significant of these changes:

- We chose to remove the phrase “We characterized the likelihood of exploitability…” in the third paragraph in Section 3.1.1 and instead state “We indicated the difficulty to produce an exploit…” This clarification was made since it is possible that other factors can impact whether a vulnerability is exploited beyond Discovery Difficulty and Exploit Difficulty, e.g., the Attacker Control or the value of the exploit target.

- We renamed Section 4 from Results to Vulnerability Types and Section 5 from Analysis of Results to Analysis of Vulnerabilities. We chose to rename these sections so that they would more clearly indicate what is being presented, i.e., the overarching types of vulnerabilities we identified through our qualitative analysis and comparisons between these types over the other variables coded.

- Because Section 5.2 regularly switches between discussing our exploitability codes (Discovery Difficulty and Exploit Difficulty) and whether teams actually submitted breaks against each vulnerability (therefore finding and exploiting it), potentially confusing the reader, we added text to clarify when each was being discussed. To differentiate between two, we referred to the former as rated exploitability and the later as actual exploitability.

- We added references to two recent papers by Naiakshina et al. [56, 58] into the controlled experiments with developers paragraph of 7. These papers both show similar results that users struggle to use APIs for secure password storage.

- Due to space limits, we removed the bar chart in Section 4 displaying the percentage of projects that introduced each vulnerability type, grouped by problem (originally Figure 1). This figure provided little additional information beyond that given in Table 2.

- Due to space limits, we removed the code listing in Section 4 that showed SC-68’s used a hardcoded string as the IV (originally Listing 1). This listing did not provide any additional context about the use of a fixed IV that could not be gleaned from the text.