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.

2 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.

2.1 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.[1] 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.[2] 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.[3] 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.

[2] Hence, our use of the term "coding" refers to a type of structured categorization for data analysis, not a synonym for programming.

[3] The last column indicates Krippendorff's α statistic [39], which we discuss in Section 3.2.

Variable	Levels	Description	Alpha [39]
Type	(See Table 2)	What caused the vulnerability to be introduced	0.85, 0.82
Attacker Control	Full / Partial	What amount of the data is impacted by an exploit	0.82
Discovery Difficulty	Execution / Source / Deep Insight	What level of sophistication would an attacker need to find the vulnerability	0.80
Exploit Difficulty	Single step / Few steps / Many steps / Probabilistic	How hard would it be for an attacker to exploit the vulnerability once discovered	1

Table 1: Summary of the vulnerability codebook.

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.

3.1.2 Project codebook

To understand the reasons teams introduced certain types of vulnerabilities, we coded several project features as well. We tracked several objective features including the lines of code (LoC) as an estimate of project complexity; the IEEE programming-language rankings [1] as an estimate of language maturity (*Popularity*); and whether the team included test cases as an indication of whether the team spent time auditing their project.

We also attempted to code projects more qualitatively. For example, the variable *Minimal Trusted Code* assessed whether the security-relevant functionality was implemented in single location, or whether it was duplicated (unnecessarily) throughout the codebase. We included this variable to understand whether adherence to security development best practices had an effect on the vulnerabilities introduced [14, pg. 32-36]. The remaining variables we coded (most of which don't feature in our forthcoming analysis) are discussed in Appendix B.

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 during the competition, sampling evenly from each. Doing so further ensured that the distribution of projects we analyzed matched the contest-break distribution in the whole set.

One assumption of our procedure was that the frequency of breaks submitted by break-it teams matches the frequency of vulnerabilities actually present in the projects. We could not sample based on the latter, because we did not have ground truth at the outset; only after analyzing the projects ourselves could we know the vulnerabilities that might have been missed. However, we can check this assumption after the fact. To do so, we performed a Spearman rank correlation test to compare the number of breaks and vulnerabilities introduced in each project [80, pg. 508]. Correlation, according to this test, indicates that if one project had more contest breaks than another, it would also have more vulnerabilities, i.e., be ranked higher according to both variables. We observed that there was statistically significant correlation between the number of breaks identified and the underlying number of vulnerabilities introduced ($\rho = 0.70$, $p < 0.001$). Further, according to Cohen's standard, this correlation is "large," as ρ is above 0.50 [23]. As a result, we are confident that our sampling procedure, as hoped, obtained a good representation of the overall dataset.

We note that an average of 27 teams per competition, plus two researchers, examined each project to identify vulnerabilities. We expect that this high number of reviewers, as well as the researchers' security expertise and intimate knowledge of the problem specifications, allowed us to identify the majority of vulnerabilities.

3.2.2 Coding

To develop our codebooks, two researchers first cooperatively examined 11 projects. For each, they reviewed associated breaks and independently audited the project for vulnerabilities. They met and discussed their reviews (totaling 42 vulnerabilities) to establish the initial codebook.

At this point, one of the two original researchers and a third researcher independently coded breaks in rounds of approximately 30 each, and again independently audited projects' unidentified vulnerabilities. After each round, the researchers met, discussed cases where their codes differed, reached a consensus, and updated the 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 alpha 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 `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

			Secure log		Secure communication		Multiuser database		Totals[3]	
Type	**Sub-type**	**Issue**	**P=52[1]**	**V=53[2]**	**P=27**	**V=64**	**P=15**	**V=65**	**P=94**	**V=182**
No Impl.	*All Intuitive*	No encryption	3 (6%)	3 (6%)	2 (7%)	2 (3%)	–	–	5 (6%)	5 (4%)
		No access control	–	–	–	–	0 (0%)	0 (0%)	0 (0%)	0 (0%)
		Total	3 (6%)	3 (6%)	2 (7%)	2 (3%)	–	–	5 (6%)	5 (4%)
	Some Intuitive	Missing some access control	–	–	–	–	10 (67%)	18 (28%)	10 (67%)	18 (10%)
		Total	–	–	–	–	10 (67%)	18 (28%)	10 (67%)	18 (10%)
	Unintuitive	No MAC	16 (31%)	16 (30%)	7 (26%)	7 (11%)	–	–	23 (29%)	23 (20%)
		Side-channel attack	–	–	11 (41%)	11 (17%)	4 (15%)	4 (6%)	15 (36%)	15 (12%)
		No replay check	–	–	7 (26%)	7 (11%)	–	–	7 (26%)	7 (11%)
		No recursive delegation check	–	–	–	–	4 (27%)	4 (6%)	4 (27%)	4 (6%)
		Total	16 (31%)	16 (30%)	18 (67%)	25 (39%)	8 (53%)	8 (12%)	42 (45%)	49 (27%)
	Total	–	17 (33%)	19 (36%)	18 (67%)	27 (42%)	12 (80%)	26 (40%)	47 (50%)	72 (40%)
Misund.	*Bad Choice*	Unkeyed function	6 (12%)	6 (11%)	2 (7%)	2 (3%)	–	–	8 (9%)	8 (4%)
		Weak crypto	4 (8%)	5 (9%)	0 (0%)	0 (0%)	–	–	4 (5%)	5 (4%)
		Homemade crypto	2 (4%)	2 (4%)	0 (0%)	0 (0%)	–	–	2 (3%)	2 (2%)
		Weak AC design	–	–	–	–	5 (33%)	6 (9%)	5 (33%)	6 (9%)
		Memory corruption	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
		Total	13 (25%)	14 (26%)	2 (7%)	2 (3%)	5 (33%)	6 (9%)	20 (21%)	22 (12%)
	Conceptual Error	Fixed value	12 (23%)	12 (23%)	6 (22%)	6 (9%)	8 (53%)	8 (12%)	26 (28%)	26 (14%)
		Insufficient randomness	2 (4%)	3 (6%)	5 (19%)	5 (8%)	0 (0%)	0 (0%)	7 (7%)	8 (4%)
		Security on subset of data	3 (6%)	3 (6%)	6 (22%)	7 (11%)	0 (0%)	0 (0%)	9 (10%)	10 (5%)
		Library cannot handle input	0 (0%)	0 (0%)	1 (4%)	1 (2%)	2 (13%)	2 (3%)	3 (3%)	3 (2%)
		Disabled protections	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
		Resource exhaustion	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (7%)	1 (2%)	1 (1%)	1 (1%)
		Total	17 (33%)	19 (36%)	15 (56%)	19 (30%)	9 (60%)	11 (17%)	41 (44%)	49 (27%)
	Total	–	28 (54%)	33 (62%)	15 (56%)	21 (33%)	10 (67%)	17 (26%)	53 (56%)	71 (39%)
Mistake	–	Insufficient error checking	0 (0%)	0 (0%)	8 (30%)	8 (12%)	4 (27%)	4 (6%)	12 (13%)	12 (7%)
		Uncaught runtime error	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	8 (12%)	5 (5%)	9 (5%)
		Control flow mistake	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	9 (14%)	5 (5%)	10 (6%)
		Skipped algorithmic step	0 (0%)	0 (0%)	4 (15%)	6 (9%)	1 (2%)	1 (2%)	5 (5%)	7 (4%)
		Null write	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
	Total	–	1 (2%)	1 (2%)	11 (41%)	16 (25%)	8 (53%)	22 (34%)	20 (21%)	39 (21%)

[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

Table 2: Number of vulnerabilities for each issue and the number of projects each vulnerability was introduced in.

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 an 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).

```
1  var nextNonce uint64 = 1337
2  ...
3  func sendMessage(conn *net.Conn, message
4    []byte) (err error) {
5    var box []byte
6    var nonce [24]byte
7
8    byteOrder.PutUint64(nonce[:], nextNonce)
9    box = secretbox.Seal(box, message, &nonce,
10     &sharedSecret)
11   var packet = Packet{Size: uint64(len(box)),
12     Nonce: nextNonce}
13   nextNonce++
14   writer := *conn
15   err = binary.Write(writer, byteOrder, packet)
16   ...
17 }
```

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

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 vulnerability instead of a *Mistake* because the teams violate the library developers' assumption as opposed to making a mistake in their code.)

```
1  self.db = self.sql.connect(filename, timeout=30)
2  self.db.execute('pragma key="' + token + '";')
3  self.db.execute('PRAGMA kdf_iter='
4    + str(Utils.KDF_ITER) + ';')
5  self.db.execute('PRAGMA cipher_use_MAC = OFF;')
6  ...
```

Listing 2: SL-22 disabled automatic MAC in SQLCipher library.

Finally, one team simply disabled protections provided transparently by the library (P=1, V=1). Team SL-22 used the SQLCipher library to implement their log as an SQL database. The library provides encryption and integrity checks in the background, abstracting these requirements from the developer. Listing 2 shows the code they used to initialize the database. Unfortunately, on line 5, they explicitly disabled the automatic MAC.

4.3 *Mistake*

Finally, some teams attempted to implement the solution correctly, but made a mistake that led to a vulnerability. The mistake type is composed of five sub-types. Some teams did not properly handle errors putting the program into an observably bad state (causing it to be hung or crash). This included not having sufficient checks to avoid a hung state, e.g., infinite loop while checking the delegation chain in the MD problem, not catching a runtime error causing the program to crash (P=5, V=9), or allowing a pointer with a null value to be written to, causing a program crash and potential exploitation (P=1, V=1).

```
1  def checkReplay(nonce,timestamp):
2      #First we check for tiemstamp delta
3      dateTimeStamp = datetime.strptime(timestamp,
4        '%Y-%m-%d %H:%M:%S.%f')
5      deltaTime = datetime.utcnow() - dateTimeStamp
6      if deltaTime.seconds > MAX_DELAY:
7          raise Exception("ERROR:Expired nonce ")
8      #The we check if it is in the table
9      global bank
10     if (nonce in bank.nonceData):
11         raise Exception("ERROR:Reinyected package")
```

Listing 3: SC-80 forgot to save the nonce.

Other mistakes led to logically incorrect execution behaviors. This included mistakes related to the control flow logic (P=5, V=10) or skipping steps in the algorithm entirely. Listing 3 shows an example of SC-80 forgetting a necessary step in the algorithm. On line 10, they check to see if the nonce was seen in the list of previous nonces (`bank.nonceData`) and raise an exception indicating a replay attack. Unfortunately, they never add the new nonce into bank.nonceData, so the check on line 10 always returns true.

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 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, $\phi \geq 0.1$ represents a small effect, ≥ 0.3 a medium effect, and ≥ 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* ($\phi = 0.29$, $p < 0.001$) and *Misunderstanding* ($\phi = 0.35$, $p < 0.001$)—were significantly more likely (roughly medium effect size) to be introduced than vulnerabilities caused by programming *Mistake*s. We observed no significant difference between *No Implementation* and *Misunderstanding* ($\phi = 0.05$, $p = 0.46$). 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* ($\phi = 0.44$, $p < 0.001$) or *Some Intuitive* ($\phi = 0.37$, $p < 0.001$) counterparts. The two more intuitive sub-types did not significantly differ ($\phi = 0.08$, $p = 0.32$) 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

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	MD	**6.68**	**[2.90, 15.37]**	**< 0.001***
	SL	**0.06**	**[0.01, 0.43]**	**0.006***
Min Trust	False	–	–	–
	True	**0.36**	**[0.17, 0.76]**	**0.007***
Popularity	C (91.5)	**1.09**	**[1.02, 1.15]**	**0.009***
LoC	1274.81	**0.99**	**[0.99, 0.99]**	**0.006***

*Significant effect – Base case (Log Estimate defined as 1)

Table 3: Summary of regression over *Mistake* vulnerabilities. Pseudo R^2 measures for this model were 0.47 (McFadden) and 0.72 (Nagelkerke).

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 *Popularity*[5] ($p = 0.009$). 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

[4] We 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.

[5] The 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.

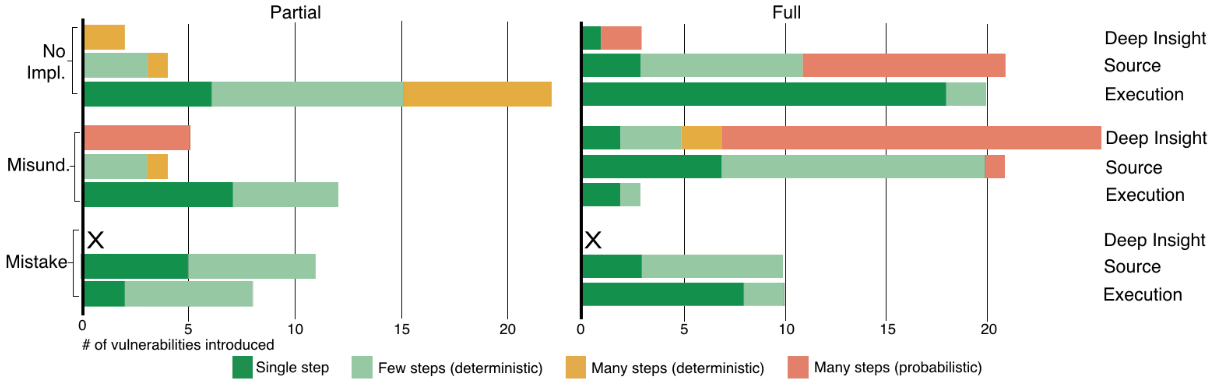

Figure 1: # vulnerabilities introduced for each type divided by *Discovery Difficulty*, *Exploit Difficulty* and *Attacker Control*.

they used. As such, we rated *Misunderstanding* vulnerabilities as hard to find significantly more often than both *No Implementation* ($\phi = 0.52$, $p < 0.001$) and *Mistake* ($\phi = 0.30$, $p = 0.02$) 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 commonType 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 ($\phi = 0.38$, $p = 0.003$), but (likely due to sample size) the difference between *Unintuitive* and *All Intuitive* is not ($\phi = 0.17$, $p = 0.17$).

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* ($\phi = 0.43$, $p = 0.001$) and *Misunderstanding* ($\phi = 0.51$, $p < 0.001$) 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* ($\phi = 0.35$, $p = 0.001$) or *No Implementation* ($\phi = 0.28$, $p = 0.006$). 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 studies 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 testers 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 non-experts (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.

[6] https://golang.org/pkg/crypto/tls/#Listen and https://www.openssl.org/docs/manmaster/man3/SSL_new.html

[7] https://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, Sliwerski 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

[1] IEEE spectrum: The top programming languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018, 2018.

[2] Coursera cybersecurity specialization. https://www.coursera.org/specializations/cyber-security, 2019.

[3] R. Abu-Salma, M. A. Sasse, J. Bonneau, A. Danilova, A. Naiakshina, and M. Smith. Obstacles to the adoption of secure communication tools. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 137–153, May 2017.

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

[5] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 289–305. IEEE, 2016.

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

[7] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 301–306, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Philippe Arteau, Andrey Loskutov, Juan Dodero, and Kengo Toda. Spotbugs. https://spotbugs.github.io/, 2019.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[10] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 281–296, Baltimore, MD, August 2018. USENIX Association.

[11] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, pages 97–106, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Dejan Baca, Bengt Carlsson, Kai Petersen, and Lars Lundberg. Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience*, 43(3):259–279, 2013.

[13] Yoav Benjamini and Yosef Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[14] Diana Burley, Matt Bishop, Scott Buck, Joseph J. Ekstrom, Lynn Futcher, David Gibson, Elizabeth K. Hawthorne, Siddharth Kaza, Yair Levy, Herbert Mattord, and Allen Parrish. Curriculum guidelines for post-secondary degree programs in cybersecurity. Technical report, ACM, IEEE, AIS, and IFIP, 12 2017.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

[16] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 459–465. Springer Berlin Heidelberg, 2011.

[17] A Colin Cameron and Pravin K Trivedi. *Regression analysis of count data*, volume 53. Cambridge university press, 2013.

[18] Ryan Camille. Computer and internet use in the united states:2016. https://www.census.gov/library/publications/2018/acs/acs-39.html, 2018.

[19] Center for Cyber Safety and Education. Global information security workforce study. Technical report, Center for Cyber Safety and Education, Clearwater, FL, 2017.

[20] Pravir Chandra. Software assurance maturity model. Technical report, Open Web Application Security Project, 04 2017.

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

[22] Steve Christey and Robert A Martin. Vulnerability type distributions in cve. `https://cwe.mitre.org/documents/vuln-trends/index.html`, 2007.

[23] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[24] Harald Cramér. *Mathematical methods of statistics (PMS-9)*, volume 9. Princeton university press, 2016.

[25] Felix Dörre and Vladimir Klebanov. Practical detection of entropy loss in pseudo-random number generators. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 678–689, 2016.

[26] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ESSoS'13, pages 197–212, Berlin, Heidelberg, 2013. Springer-Verlag.

[28] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[29] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, 2014.

[30] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[31] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *USENIX conference on Web application development*, 2011.

[32] FIRST.org. Common vulnerability scoring system. `https://www.first.org/cvss/calculator/3.0`, 2016. (Accessed 12-19-2016).

[33] Ronald A Fisher. On the interpretation of χ 2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

[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.

[35] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[36] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[37] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. "we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 357–373, Baltimore, MD, August 2018. USENIX Association.

[38] William R. Harris, Somesh Jha, Thomas W. Reps, and Sanjit A. Seshia. Program synthesis for interactive-security systems. *Form. Methods Syst. Des.*, 51(2):362–394, November 2017.

[39] Andrew F Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89, 2007.

[40] Mariana Hentea, Harpal S Dhillon, and Manpreet Dhillon. Towards changes in information security education. *Journal of Information Technology Education: Research*, 5:221–233, 2006.

[41] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[42] Melanie Jones. Why cybersecurity education matters. https://www.itproportal.com/features/why-cybersecurity-education-matters/, 2019.

[43] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, 2012.

[44] Nick Kolakowski. Software developer jobs will increase through 2026. https://insights.dice.com/2019/01/03/software-developer-jobs-increase-2026/, 2019.

[45] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[46] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 7. ACM, 2014.

[47] Timothy C Lethbridge, Jorge Diaz-Herrera, Richard Jr J LeBlanc, and J Barrie Thompson. Improving software practice through education: Challenges and future trends. In *Future of Software Engineering*, pages 12–28. IEEE Computer Society, 2007.

[48] Gary McGraw, Sammy Migues, and Brian Chess. Software security framework | bsimm, 2009. (Accessed 05-22-2018).

[49] Gary McGraw and John Steven. Software [in]security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal. http://www.informit.com/articles/article.aspx?p=1680863, 2011. (Accessed 02-26-2017).

[50] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74, Oct 2013.

[51] Andrew Meneely, Alberto C Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44. ACM, 2014.

[52] Andrew Meneely and Oluyinka Williams. Interactive churn metrics: socio-technical variants of code churn. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–6, 2012.

[53] Microsoft. Microsoft security development lifecycle practices. https://www.microsoft.com/en-us/securityengineering/sdl/practices, 2019.

[54] Mitre. Cve. https://cve.mitre.org/, 2019.

[55] Mitre. Cwe: Common weakness enumeration. https://cwe.mitre.org/data/definitions/1000.html/, 2019.

[56] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. &#34;if you want, i can store the encrypted password&#34;: A password-storage field study with freelance developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 140:1–140:12, New York, NY, USA, 2019. ACM.

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

[58] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 297–313, Baltimore, MD, 2018. USENIX Association.

[59] William Newhouse, Stephanie Keith, Benjamin Scribner, and Greg Witte. Nist special publication 800-181, the nice cybersecurity workforce framework. Technical report, National Institute of Standards and Technology, 08 2017.

[60] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 315–328, Baltimore, MD, 2018. USENIX Association.

[61] OWASP. Top 10-2017 top 10. https://www.owasp.org/index.php/Top_10-2017_Top_10, 2017.

[62] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 426–437, New York, NY, USA, 2015. ACM.

[63] Adrian E Raftery. Bayesian model selection in social research. *Sociological methodology*, pages 111–163, 1995.

[64] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications in the developing world. In *USENIX Security Symposium*, pages 17–32, 2015.

[65] Tony Rice, Josh Brown-White, Tania Skinner, Nick Ozmore, Nazira Carlage, Wendy Poland, Eric Heitzman, and Danny Dhillon. Fundamental practices for secure software development. Technical report, Software Assurance Forum for Excellence in Code, 04 2018.

[66] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 690–703, New York, NY, USA, 2016. ACM.

[67] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[68] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[69] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: A controlled experiment. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 451–460. IEEE, 2013.

[70] K Serebryany. libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2015.

[71] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[72] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS '17. ACM, 2017.

[73] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[74] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*, volume 15. Newbury Park, CA: Sage, 1990.

[75] Larry Suto. Analyzing the effectiveness and coverage of web application security scanners. Technical report, BeyondTrust, Inc, 2007.

[76] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. Technical report, BeyondTrust, Inc, 2010.

[77] Patrick Thibodeau. India to overtake u.s. on number of developers by 2017. https://www.computerworld.com/article/2483690/it-careers/india-to-overtake-u-s--on-number-of-developers-by-2017.html, 2013.

[78] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 262:1–262:12, New York, NY, USA, 2018. ACM.

[79] Daniel Votipka, Rock Stevens, Elissa M Redmiles, Jeremy Hu, and Michelle L Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. *Proc. of the IEEE*, 2018.

[80] Arnold D Well and Jerome L Myers. *Research design & statistical analysis*. Psychology Press, 2nd edition, 2003.

[81] Michal Zalewski. American fuzzing lop (afl). http://lcamtuf.coredump.cx/afl/, 2014.

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.

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

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

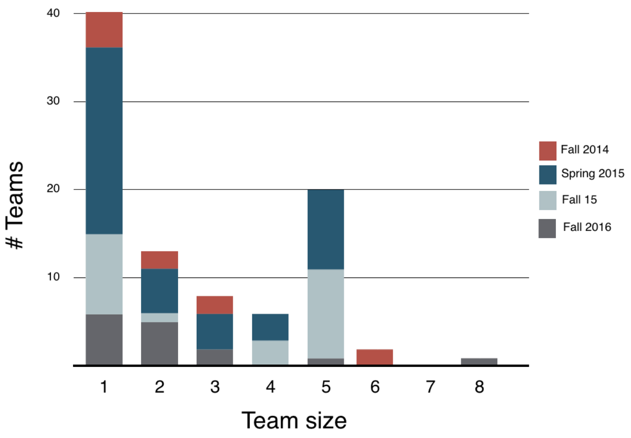

Figure 2: Histogram of team size split by competition.

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

Variable	Levels	Description	Alpha
Modular	T / F	Whether the project is segmented into a set of functions and classes each performing small subcomponents of the project	1
Variable Naming	T / F	Whether the author used variable names that indicated the purpose of the variable	1
Whitespace	T / F	Whether the author used whitespace (i.e., indentation and new lines) to allow the reader to easily infer control-flow and variable scope	1
Comments	T / F	Whether the author included comments to explain blocks of the project	0.89
Economy of Mechanism	T / F	How complicated are the implementations of security relevant functions	0.84
Minimal Trusted Code	T / F	Whether security relevant functions are implemented once or re-implemented each time they are needed	0.84

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.

Variable	Value	Log Estimate	CI	p-value
Popularity	C (91.5)	1.03	[0.98, 1.09]	0.23

*Significant effect – Base case (Estimate=1, by definition)

Table 6: Summary of regression over *Bad Choice* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.03 (Nagelkerke).

C.2 Model Selection

We calculated the Bayseian 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.

Variable	Value	Log Estimate	CI	p-value
MOOC	False	–	–	–
	True	1.76	[0.70, 4.34]	0.23

*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).

Variable	Value	Log Estimate	CI	p-value
Yrs. Experience	8.9	1.12	[0.82, 1.55]	0.47

*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).

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	SL	1.02	[0.98, 1.07]	0.373

*Significant effect **–** Base case (Estimate=1, by definition)

Table 9: Summary of regression over *Some Intuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.07 (Nagelkerke).

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	MD	0.58	[0.25, 1.35]	0.21
	SL	**0.31**	**[0.15, 0.60]**	**< 0.001***

*Significant effect **–** Base case (Estimate=1, by definition)

Table 10: Summary of regression over *Unintuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.07 (McFadden) and 0.16 (Nagelkerke).