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

Daniel Votipka, Kelsey 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 programmers to consider many possible threats and mitigations. This paper investigates how and why programmers, despite having a baseline of security experience, make security-relevant errors. To do this, we conducted an in-depth analysis of 76 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 against it (including vulnerabilities we identified ourselves). We labeled vulnerabilities by type, severity, and ease of exploitation, and projects according to security implementation strategy. Several patterns emerged. For example, we found that simple mistakes were least common: only 26% of projects introduced such an error. Conversely, vulnerabilities arising from a misunderstanding of security concepts were significantly more common: 84% of projects introduced at least one such error. Overall, our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

1 Introduction

Producing secure programs is a difficult task. Despite continued improvements in automated vulnerability discovery tools [5, 8, 9, 22, 41, 54, 58, 61, 62] and expansion in security education [16, 34, 36, 40], many vulnerabilities are discovered every year in production code [17, 18, 45]. NIST's National Initiative for Cybersecurity Education framework highlights the size of the challenge developers face, outlining 44 distinct areas of knowledge necessary for secure development [48]. Further, even in NIST's in-depth specification, many of the knowledge areas are very broad, such as knowledge of "cyber threats and vulnerabilities" or "secure coding techniques."

Unfortunately, with the increasing pervasiveness of computing and the rising number of professional developers [15, 38, 63], it is not reasonable to expect all developers to attain this level of security expertise. Developers' limited time and attention — e.g., for security training, or for learning about and using new programming and testing tools — implies the importance of identifying the most critical and effective security interventions to prioritize. In practice, the effectiveness (or lack thereof) of any particular intervention is not simply about the intervention itself, but also about its relation to how and why developers make security errors. That is, an efficient-on-paper tool that does not reflect real developer needs may not be effective in practice.

In this paper, we make progress on this issue by investigating how and why programmers, despite a baseline of security experience, make security-relevant errors. To do this, we carried out a systematic, in-depth examination (using best practices developed for qualitative assessments) of vulnerabilities present in 76 projects sampled from submissions to the *Build it, Break it, Fix it*[1] (BIBIFI) secure-coding competition series [53]. In each competition, participating teams (many of which were enrolled in a security MOOC) first developed programs for a particular scenario, 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. Upon completion, 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 172 unique security vulnerabilities associated with those projects.

By studying code produced within security competitions, we can identify patterns in how different teams make security errors when approaching the same problem specification, enabling insights that are difficult to obtain when examining

[1] https://builditbreakit.org

real-world programs with varying goals and requirements. At the same time, the contest format — in which teams had several weeks to build their project submissions, using any languages or tools they preferred — provides more ecological validity than lab studies in which developers complete small, highly specified programming tasks.

Our rigorous manual analysis of this dataset identified several interesting trends, with implications for improving secure-development training, security-relevant APIs [2, 31, 46], 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 26% of projects introduced such an error. Mitigations to these sorts of mistakes are plentiful. Analyzing the data, we found that minimizing the trusted code base (e.g., by avoiding duplication of critical security functionality) led to significantly fewer mistakes. Moreover, we believe that modern analysis tools and testing techniques [6, 7, 12, 13, 21, 25, 32, 35, 37, 56, 57, 65] should uncover many of them. All but one were found and exploited by opposing teams. In short, this class of bugs appears to be both relatively uncommon and amenable to existing tools and best practices, suggesting it can be effectively managed.

Vulnerabilities arising from misunderstanding of security concepts were significantly more common: 84% of projects introduced at least one such error. These misunderstandings sometimes led teams to miss unintuitive security requirements altogether (50% of projects); e.g., several teams used encryption to protect confidentiality but failed to also use MACs to protect integrity. Teams also often made conceptual errors in attempting to apply a security mechanism (49% of projects); for example, several projects failed to provide randomness when an API expects it. Although these vulnerabilities were common, they proved difficult to exploit: only 76% were exploited by other teams (compared to 97% of simple mistakes), and our qualitative labeling identified 32% 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. Perhaps there is a role for BIBIFI itself in education: the experience of having your code exploited may serve as a more robust lesson than simply hearing about a potential problem.

2 Data

This section presents the Build It, Break It, Fix It (BIBIFI) secure-programming competition [53], the 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.[2] 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. 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 asked teams to implement two programs: one to securely append records to a log, and one query the log's contents. Teams had to protect against an adversary with access to the log and ability to modify it, but who lacked a so-called "authentication token." Teams were expected (but not told explicitly) to utilize cryptographic functions to both encrypt the log and protect its integrity. The

[2] Source code obfuscation was against the rules. Complaints of violations were judged by contest organizers.

build-it score was measured by log query/append latency and space utilization, and teams could implement several optional features. Security breaks could demonstrate compromises to either integrity or confidentiality on sample logs generated by the organizers during the break-it phase, for each project.

Secure communication (SC, Fall 2015, RI size: 1,124 lines of Haskell). This problem asked teams to build a pair of client/server programs. These represented a bank and an ATM, which could initiate account transactions (e.g., account creation, deposits, withdrawals, etc.). Teams needed to protect bank data integrity and confidentiality despite an adversary acting as a man-in-the-middle (MITM), with the ability to read and manipulate communications between the two parties. Once again, build teams needed to use cryptographic functions properly, and to consider challenges such as replay attacks, server availability, and side-channel data leakage. Build-it performance was measured by transaction latency (there were no optional features). Break teams provided a script and a MITM program to demonstrate exploitations violating confidentiality or integrity of bank data.

Multiuser database (MD, Fall 2016, RI size: 1,080 lines of OCaml). This problem asked teams to create a server that ran programs consisting of a series of commands in a domain-specific language to store and process key-value data. Each command set was submitted by a user whose access to the server's data was constricted by access control rules defined by prior commands. Teams were tasked with correctly implementing commands that executed queries and updates to server values while properly enforcing access control rules involving read, write, and delegate privileges. Build-it performance was assessed by program running time, and the project offered several optional features. The attacker in this scenario was able to submit arbitrary commands to the server. Breaks were demonstrated as test cases that caused security-relevant deviations from the behavior of the RI.

Project Characteristics. Teams used a variety of languages in their projects. Python was most popular overall (33 teams, 43%), with Java also widely used (15, 20%), and C/C++ third (6 each). Other languages used by at least one team include Ruby, Perl, Go, Haskell, Scala, PHP, JavaScript and Visual Basic. For the secure log problem, projects ranged from 149 to 3397 lines of code (median 1007). 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. 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).[3] 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 median of 12 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 [3, 4].

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.

- We only examine three secure-development scenarios. Focusing on a small set of problems, with many teams performing the same task, allows us to gain deep insight into how teams approach these problems, but may not generalize to other security-critical tasks. While not comprehensive, these three problems were selected to exercise important real-world requirements like secure communication and access control.

- In some larger companies, developers are supported by teams of security experts [64] who provide design suggestions and set requirements (or preferences) for secure libraries and practices. We do not provide teams with any security recommendations for design or implementation of their project (though MOOC students did receive relevant training). While this is not representative of all development settings, it allows

[3] https://www.coursera.org/specializations/cyber-security

us to measure the behaviors of the developers themselves without the confounding effects of the security team.

• In the contest setting, teams develop their codebase from scratch. While this is an important phase of development, it is not the only one. For example, a developer might be tasked with adding functionality to a pre-existing codebase. Having to integrate with a previously developed—and possibly vulnerable—codebase could affect the number and type of vulnerabilities introduced. Having teams implement code from scratch allows us to focus our analysis on the initial phase of design and implementation (of a whole program, or a program component), but further research should examine other settings.

• 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 [46, 47]. 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, 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 *severe* are the vulnerabilities? If exploited, what is the effect on the system?

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 a sample of 76 BIBIFI projects (54% of all BIBIFI projects) and the 866 breaks submitted against them. We performed a rigorous *iterative open coding* [60, 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.[4] The collection of labels is called a *codebook*. The ultimate codebook we developed provides labels for vulnerabilities—their type, severity, 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. Following standard terminology, we refer to these labels as *variables* and their possible values as *levels*. The structure of our vulnerability codebook is given in Table 1;[5] we discuss the project codebook below.

3.1.1 Vulnerability codebook

To measure the types of vulnerabilities in each project, we characterized them across three variables.

Vulnerability type. The *Type* variable characterizes the underlying source of a vulnerability (RQ1). For example, a vulnerability in which encryption initialization vectors (IVs) are reused is classified as having the issue *insufficient randomness.* We identified more than 20 different issues; these are discussed in detail in Section 4.

Severity of exploitation. The *Severity* variable characterizes the impact of a vulnerability's exploitation (RQ2) as either a full compromise 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 characterized the likelihood of exploitabilty (RQ3) using two variables, *Discovery Difficulty* and *Exploit Difficulty*. The first characterizes the amount of knowledge the attacker must have to 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, determined by two questions: First, are many steps required to move the project into an exploitable state, or just one

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

[5] The last column indicates Krippendorff's α statistic [33], which we discuss in Section 3.2.

Variable	Levels	Description	Alpha [33]
Type	(See Table 2)	What caused the vulnerability to be introduced	0.85, 0.82
Severity	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.

or a few? If many steps are required, is the number of them probabilistic? 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. For brevity, we only mention one variable that was found to have an effect on the vulnerabilities introduced, *Minimal Trusted Code*. This variable assesses 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 [11, pg. 32-36].

In addition to qualitative codes for each project, we tracked several objective features including the lines of code as an estimate of project complexity; the IEEE programming-language rankings[6] as an estimate of language maturity; and whether the team included test cases as an indication of whether the team spent time auditing their project.

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

Because of the intensive nature of our analysis, we considered a random sample of projects rather than analyzing every one. Our sampling procedure was divided into two groups: projects with submitted breaks and those without. As our primary goal was to identify the types of vulnerabilities developers introduced, we focused primarily on projects that had breaks submitted against them. For this group, we randomly sampled using the distribution of breaks per project for all projects in the competition, as opposed to a uniform distribution. We performed our weighted random sampling for each competition until at least 30% of all submitted breaks were covered. We chose a weighted random sample to ensure our final break distribution matched that of the full competition population. This prevented biasing toward more or less vulnerable samples. For the projects without breaks, we performed unweighted random sampling (i.e., using a uniform distribution) until 25% of the break-less projects were selected.

We randomly selected 56 projects targeted by at least one break (66% of projects with breaks) and 20 targeted by none (35% of break-less projects), for 76 projects in total.

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; however our results should be considered a lower bound.

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 [33]. Krippendorff's α is a conservative measure which consid-

[6] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

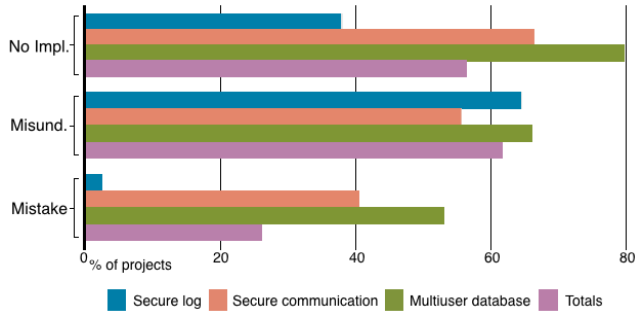

Figure 1: Percentage of projects that introduced a *Mistake*, *Misunderstanding*, and *No Implementation* vulnerability, grouped by problem

ers improvement over simply guessing. Krippendorff et al. recommend a threshold of $\alpha > 0.8$ as a sufficient level of agreement [33]. 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 16 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 Results

Our manual analysis of 76 BIBIFI projects identified 172 unique vulnerabilities. We categorized each based on our codebook into 23 different *Types*. Table 2 presents this data. Issues are organized according to three main classes: *No Implementation*, *Misunderstanding*, and *Mistake*. These were determined systematically using *axial coding*, which identifies connections between codes and extracts higher-level themes [60, 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 the type. Figure 1 additionally illustrates the percentage of projects that introduced a vulnerability of each class, grouped by problem.

This section presents, for each class, detailed descriptions and examples. The next section analyzes these results, considering vulnerability class prevalence, severity, and likelihood to be exploited.

4.1 *No Implementation*

A vulnerability type was classed as *No Implementation* when a team failed to even attempt to implement a necessary security mechanism, presumably because they did not realize it was needed. This class is further divided into the sub-classes *All Intuitive*, *Some Intuitive*, and *Unintuitive*. In the first two categories 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* category was used if the security requirement was not directly stated or was otherwise unintuitive (e.g., using MAC to provide integrity [1]).

All Intuitive vulnerabilities had two subclasses: 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* subclass 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 `admin`-only commands. For *Unintuitive* vulnerabilities, there were four issues: teams failed to include a MAC to protect data integrity in the secure log (P=12, V=12) and secure communication (P=7, V=7) problems; prevent side-channel data leakage through packet sizes or success/failure responses (P=11, V=11) in the secure communication and multiuser database problems; 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 classed as *Misunderstanding* when a team attempted to implement a security mechanism, but failed due to a conceptual misunderstanding. We sub-classed these as either *Bad Choice* or *Conceptual Error*.

4.2.1 *Bad Choice*

Five issues fall under this sub-class, 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 deterministic algorithms that should have employed a key (P=7, V=7), weak algorithms (P=3, V=4), or homemade encryption (P=2, V=2).[7] For example, SL-69[8] 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 default delegation improperly. Instead of delegating rights to a new user at the

[7] A dash is used to indicate that a vulnerability does not apply to a problem.

[8] We identify projects using a shortened version of the problem and a randomly assigned ID.

Class	Sub-class	Issue	Secure log		Secure communication		Multiuser database		Totals[3]	
			P=34[1]	V=43[2]	P=27	V=64	P=15	V=65	P=76	V=172
No Impl.	*All Intuitive*	No encryption	3 (9%)	3 (7%)	2 (7%)	2 (3%)	–	–	5 (8%)	5 (5%)
		No access control	–	–	–	–	0 (0%)	0 (0%)	0 (0%)	0 (0%)
		Total	3 (9%)	3 (7%)	2 (7%)	2 (3%)	–	–	5 (8%)	5 (5%)
	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	12 (35%)	12 (28%)	7 (26%)	7 (11%)	–	–	19 (31%)	19 (18%)
		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	12 (35%)	12 (28%)	18 (67%)	25 (39%)	8 (53%)	8 (12%)	38 (50%)	45 (26%)
	Total	–	13 (48%)	15 (35%)	18 (67%)	27 (42%)	12 (80%)	26 (40%)	43 (57%)	68 (40%)
Misund.	*Bad Choice*	Unkeyed function	5 (15%)	5 (11%)	2 (7%)	2 (3%)	–	–	7 (11%)	7 (7%)
		Weak crypto	3 (9%)	4 (9%)	0 (0%)	0 (0%)	–	–	3 (5%)	4 (4%)
		Homemade crypto	2 (6%)	2 (5%)	0 (0%)	0 (0%)	–	–	2 (3%)	2 (2%)
		Weak AC design	–	–	–	–	5 (33%)	6 (9%)	5 (33%)	6 (9%)
		Memory corruption	1 (3%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
		Total	11 (32%)	12 (28%)	2 (7%)	2 (3%)	5 (33%)	6 (9%)	18 (24%)	20 (12%)
	Conceptual Error	Fixed value	8 (24%)	8 (19%)	6 (22%)	6 (9%)	8 (53%)	8 (12%)	22 (29%)	22 (13%)
		Insufficient randomness	2 (6%)	3 (7%)	5 (19%)	5 (8%)	0 (0%)	0 (0%)	7 (9%)	8 (5%)
		Security on subset of data	3 (9%)	3 (7%)	6 (22%)	7 (11%)	0 (0%)	0 (0%)	9 (12%)	10 (6%)
		Library cannot handle input	0 (0%)	0 (0%)	1 (4%)	1 (2%)	2 (13%)	2 (3%)	3 (4%)	3 (2%)
		Disabled protections	1 (3%)	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	13 (38%)	15 (35%)	15 (56%)	19 (30%)	9 (60%)	11 (17%)	37 (49%)	45 (26%)
	Total	–	22 (65%)	27 (63%)	15 (56%)	21 (33%)	10 (67%)	17 (26%)	47 (62%)	65 (38%)
Mistake	–	Insufficient error checking	0 (0%)	0 (0%)	8 (30%)	8 (12%)	4 (27%)	4 (6%)	12 (16%)	12 (7%)
		Uncaught runtime error	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	8 (12%)	5 (3%)	9 (5%)
		Control flow mistake	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	9 (14%)	5 (7%)	10 (6%)
		Skipped algorithmic step	0 (0%)	0 (0%)	4 (15%)	6 (9%)	1 (2%)	1 (2%)	5 (7%)	7 (4%)
		Null write	1 (3%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)
	Total	–	1 (3%)	1 (2%)	11 (41%)	16 (25%)	8 (53%)	22 (34%)	20 (26%)	39 (23%)

[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.

time of creation based on the default delegator's rights, they delegated rights at the time of resource access. Because it's possible that the default delegator received access to data between time of creation and time of use, users 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 class this as *Mistake*, we considered it a bad choice because a safe function (*strlcpy*) could have been used instead.

4.2.2 *Conceptual Error*

Teams often chose a secure high-level approach but introduced a vulnerability at implementation time due to a conceptual misunderstanding (rather than a simple mistake). This *Conceptual Error* sub-class manifested in six ways.

```
1 def fillercrypter(sharedkey, text):
2     ...
3     encryption_suite = AES.new(sharedkey,
4       AES.MODE_CBC, 'This is an IV456')
5     ...
```

Listing 1: SC-68 used a hardcoded string as the IV.

Most commonly, teams used a fixed value when an random or unpredictable one was necessary (P=22, V=22). This included using hardcoded account passwords (P=8, V=8) or encryption keys (P=2, V=2), or using a fixed IV (V=12, N=12). For example, SC-68 used a fixed IV, shown in Listing 1.

```
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 2: 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 2 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 2), 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 3: 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 3 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 class is composed of five subclasses: insufficient error checking (P=12, V = 12), uncaught runtime error (P=5,

V=9), control flow mistake (P=5, V=10), skipped algorithmic step (P=5, V=7), and null write (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 4: SC-80 forgot to save the nonce.

Listing 4 shows an example 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 Results

This section considers the results gathered in Table 2, considering for each vulnerability class prevalence, severity, and likelihood to be exploited. 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

We observed a clear trend that teams often struggled to completely understand security concepts. For this analysis, we performed planned pairwise comparisons of the number of projects containing vulnerabilities of a given class. We compare each class to each other and compare sub-classes of the same class. We use a Chi-squared test—appropriate for frequency comparisons of categorical data [29]—and adjust for multiple comparisons using a Benjamini-Hochberg (BH) correction [10]. For each test, we report effect size, calculated as the measure of association of the two variables tested (ϕ) [20, 282-283], and the statistical significance of the result (p-value). As a rule of thumb, $\phi \geq 0.1$ represents a small effect, ≥ 0.3 a medium effect, and ≥ 0.5 a large effect [19]. A p-value less than 0.05 is considered significant.

Teams often did not understand security concepts. We found that both classes of vulnerabilities relating to a lack of security knowledge—*No Implementation* ($\phi = 0.29$, $p < 0.001$) and *Misunderstanding* ($\phi = 0.34$, $p < 0.001$)—were significantly more likely (roughly medium effect size) to be introduced than vulnerabilities caused by programming mistakes. As seen in Figure 1, this overall trend held up across all problems. We observed no significant difference between *No Implementation* and *Misunderstanding*. These results indicate that efforts to address conceptual gaps should be prioritized. Focusing on these issues of understanding, we make the following observations.

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	MD	**6.64**	**[2.88, 15.27]**	**< 0.001***
	SL	**0.07**	**[0.01, 0.57]**	**0.012***
Min Trust	False	–	–	–
	True	**0.37**	**[0.18, 0.78]**	**0.008***
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 0.44 (McFadden) and 0.72 (Nagelkerke).

Unintuitive security requirements are commonly skipped. Of the *No Implementation* vulnerabilities, we found that the *Unintuitive* subclass was much more common than its *All Intuitive* ($\phi = 0.47$, $p < 0.001$) or *Some Intuitive* ($\phi = 0.38$, $p < 0.001$) counterparts. The two more intuitive subclasses did not significantly differ ($\phi = 0.09$, $p = 0.32$) This indicates that developers do attempt to provide security — at least when incentivized to do so — but struggle to consider all the unintuitive ways an adversary could attack a system. Therefore, they regularly leave out some necessary controls.

Teams often used the right security primitives, but did not know how to use them correctly. Among the *Misunderstanding* vulnerabilities, we found that the *Conceptual Error* sub-class was significantly more likely to occur than *Bad Choice* ($\phi = 0.25$, $p = .003$). 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) [14, 67-106] we performed for issues in the *Mistakes* class.[9]

[9] 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 [51]. We include further details of the initial covariates and the selection process in

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 5.79 indicates that teams were 5.79× 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 2.65× and 13.44×. 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 0.07× 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, implementing the secure components multiple times (*Minimal Trusted Code*) was associated with an 0.3× decrease in *Mistakes*, suggesting that 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 three other checks, allowing the vulnerability to remain in their project.

Mistakes are more common in popular languages. Teams that used more popular languages are expected to have a 1.08× increase in *Mistakes* for every one unit increase in popularity over the mean *Popularity*[10] ($p = 0.008$). This means, for example, that a language 5 points more popular than average would be associated with a 1.47× increase in *Mistakes*. One possible explanation is that this variable is a proxy for experience, as many participants who used less popular languages also knew more languages and were therefore 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 Effect of Exploitation

To answer RQ2 and RQ3, we compare the severity and likelihood of exploitation of the identified vulnerabilities. Figure 2 shows the number of vulnerabilities for each class with each bar divided by *Exploit Difficulty*, bars grouped by *Discovery Difficulty*, and the left and right charts showing partial and full severity vulnerabilities, respectively.

Appendix A, along with discussion of other regressions we tried but do not include for lack of space.

[10]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.

For most of the variables discussed in this section, we made the same set of planned pairwise Chi-squared tests described in Section 5.1. When comparing how many vulnerabilities were found by at least one team in the Break It phase, we conducted the same pairwise comparisons using Fisher's exact test, which is appropriate when the values of some cell values are less than five [28]. 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 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 they used. As such, we coded *Misunderstanding* vulnerabilities as hard to find significantly more often than both *No Implementation* ($\phi = 0.36$, $p < 0.001$) and *Mistake* ($\phi = 0.22$, $p = 0.05$) vulnerabilities.

Interestingly, we did not observe a significant difference between the *Misunderstanding* and *No Implementation* classes regarding whether a Break It team found the vulnerability. This indicates that even though *Misunderstanding* vulnerabilities were expected to be more difficult to find and exploit, a sufficiently large number of eyes can help close this gap.

That being said, the *Types* that were least commonly found by Break It teams were of the *Misunderstanding* class. Specifically, using a weak algorithm (Not Found=2, Found=2), and using a homemade algorithm (Not Found=1, Found=1) were only found half of the time. These vulnerabilities presented a mix of challenges, with some being 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 was difficult to read. However, once an attacker realizes that the team has essentially reimplemented the Wired Equivalent Protocol (WEP), a simple lookup in Wikipedia would reveal the exploit. Conversely, identifying that a non-random IV was used for encryption is simple, but successful exploitation can require significant time and effort.

***No Implementations* are easy to find.** Unsurprisingly, a majority of *No Implementation* vulnerabilities were coded as easy to find (V=41, 60% 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 coded as difficult to exploit; however, 42% of *Unintuitive* vulnerabilities were (V=19). The difference between *Unintuitive* and *Some Intuitive* is significant ($\phi = 0.38$, $p = 0.001$), but (likely due to sample size) the difference between *Unintuitive* and *All Intuitive* is not ($\phi = 0.27$, $p = 0.83$).

As an example, SL-7 did not use a MAC to detect modifications to their encrypted files. This was a very simple mistake

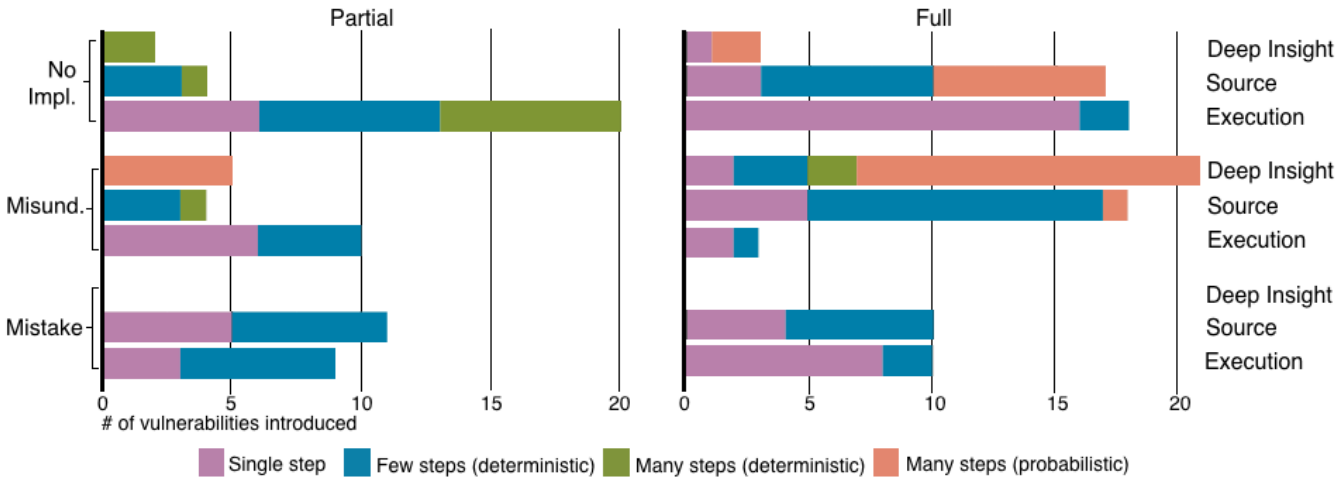

Figure 2: Number of vulnerabilities introduce for each class divided by *Discovery Difficulty*, *Exploit Difficulty* and *Severity*.

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 attacker could require a large number of tests to find a suitable modification.

***Mistakes* are easy to find and exploit.** We coded *Mistakes* as easy to exploit significantly more often than either *No Implementation* ($\phi = 0.33$, $p = 0.002$) or *Misunderstanding* ($\phi = 0.39$, $p < 0.001$) vulnerabilities. Further, all *Mistakes* were coded as easy to exploit. This suggests that, although *Mistakes* were least common, any that do find their way into production code will be found and exploited. Fortunately, code review may be sufficient to find many of these vulnerabilities: *Mistakes* were also found and exploited during the Break It phase significantly more often than either *Misunderstanding* ($\phi = 0.30$, $p = 0.02$) or *No Implementation* ($\phi = 0.25$, $p = 0.02$). In fact, only one *Mistake* (0.03%) was not found by any Break It team.

No significant difference in severity. We find no significant differences between classes or subclasses in the incidence of full and partial severity. We do observe a trend that *Misunderstanding* vulnerabilities exhibited full severity more often (V=44, 68% of *Misunderstandings*) than *No Implementation* and *Mistake* (V=40, 59% and V=20, 51%, respectively); this trend could be further investigated in future work.

6 Discussion and Recommendations

This section considers some recommendations and ideas for further research suggested by our results.

API design. Our results support the basic idea that security controls are best applied transparently, e.g., using simple APIs [31]. 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 [2, 46]. 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.[11] As another example, SL-22 (Listing 3) 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.[12] In short, library developers should be careful to highlight all security-critical assumptions.

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

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

[12] https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC

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). If education is to blame, it could be that the topics were not driven home in a sufficiently meaningful manner. Doing so is, as it turns out, a goal of the BIBIFI competition itself—failure to get it right ought to serve as a valuable lesson. It would be interesting to see how well competitors from one contest did 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 [6,35], Infer [13], and FlowDroid [7]; symbolic executors like KLEE [12] and angr [57]; fuzz testers like AFL [65] or libfuzzer [56]; and dynamic analyses like libdft [37] and TaintDroid [25] could have uncovered vulnerabilities relating to memory corruption, improper parameter use (like a fixed IV [21]), 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, e.g., program synthesis [32], could be brought to bear on these issues.

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 classes were introduced. This is consistent with prior research [49] 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). 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 [53] 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.

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 [42,44]. They found that vulnerabilities are associated with higher-than-average code churn, committing authors who are new to the codebase, and interactive churn (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 [43]. Significantly earlier, Sliwerski et al. explored mechanisms for identifying bug-fix commits in the CVS archives for Eclipse, finding, e.g., that fix-inducing changes typically span more files than other commits [59]. Perl et al. used metadata from Github and CVEs to train a classifier to identify commits that might contain vulnerabilities [50].

Other researchers have investigated trends in CVEs and the National Vulnerabilty 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 [18]. As a continuation, Chang et al. explored CVEs and the NVD from 2007–2010, showing that the percentage of high-severity vulnerabilities decreased over time, but that more than 80% of all examined vulnerabilities were exploitable via network access without authentication [17]. 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 present in the CVE database were caused by bugs in the libraries themselves, while 83% were caused by developer misuse of the libraries [39]. This accords with our *Conceptual Error* results. Egele et al. developed an analyzer to recognize specific cryptographic problems and found that nearly 88% of the applications on the Google Play Store make at least one of these mistakes when using cryptographic APIs [24]. Other researchers used fuzzing and static analysis to identify problems with SSL/TLS implementations in libraries and in Android apps [26, 30]. 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 [52]. These works examine specific categories 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 classes 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 [49]. 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 [2,46]. Perhaps closest to our work, Finifter et al. compared different teams' attempts to build a secure web application using different tools and frameworks [27]. 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 [23]. Other work suggested that users found more vulnerabilities faster with static analysis than with black-box penetration testing [55].

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 that we understand which security errors programmers tend to make, and why. With the goal of increasing this understanding, this paper has presented a systematic, qualitative study of 76 program submissions to the build it, break it, fix it secure programming contest. Each submission implemented one of three non-trivial, security-relevant programming problems. Considering the 866 exploits against these submitted programs, produced by fellow competitors, we took about six months to label 172 unique security vulnerabilities (including many we found ourselves), according to type, severity, and ease of exploitability, using used 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—almost 9 in 10 projects failed to implement a key part of a defense, or did so incorrectly, while just under 3 in 10 made simple mistakes. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

[17] 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.

[18] Steve Christey and Robert A Martin. Vulnerability type distributions in cve, 2007.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

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

[36] Melanie Jones. Why cybersecurity education matters, 2019.

[37] 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.

[38] Nick Kolakowski. Software developer jobs will increase through 2026, 2019.

[39] 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.

[40] 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.

[41] Gary McGraw and John Steven. Software [in]security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal, 2011. (Accessed 02-26-2017).

[42] 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.

[43] 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.

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

[45] Mitre. Cve, 2019.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

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

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

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

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

[63] Patrick Thibodeau. India to overtake u.s. on number of developers by 2017, 2013.

[64] 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.

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

A Regression Analysis

For each subclass of vulnerability types, we performed a poisson regression [14, 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 the main paper due to their lack of significant results or poor model fit.

A.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.

Variable	Value	Log Estimate	CI	p-value
Problem	SC	–	–	–
	MD	**5.40**	**[1.09, 26.76]**	**0.039***
	SL	**4.76**	**[1.07, 21.29]**	**0.041***

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

Table 4: Summary of regression over *Bad Choice* vulnerabilities. Pseudo R^2 measures for this model were 0.07 (McFadden) and 0.12 (Nagelkerke).

Variable	Value	Log Estimate	CI	p-value
Security Training	False	–	–	–
	True	2.13	[0.84, 5.41]	0.11

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

Table 5: Summary of regression over *Conceptual Error* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.05 (Nagelkerke).

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.

A.2 Model Selection

We calculated the Bayseian Information Criterion (BIC)—a standard metric for model fit [51]—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 [19]. 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.

A.3 Results

Tables 4–8 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
Problem	SC	–	–	–
	SL	1.19	[0.20, 7.13]	0.848

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

Table 6: Summary of regression over *All Intuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.001 (McFadden) and 0.001 (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 7: 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	[0.26, 1.28]	0.174
	SL	**0.38**	**[0.19, 0.76]**	**0.006***

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

Table 8: Summary of regression over *Unintuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.06 (McFadden) and 0.12 (Nagelkerke).