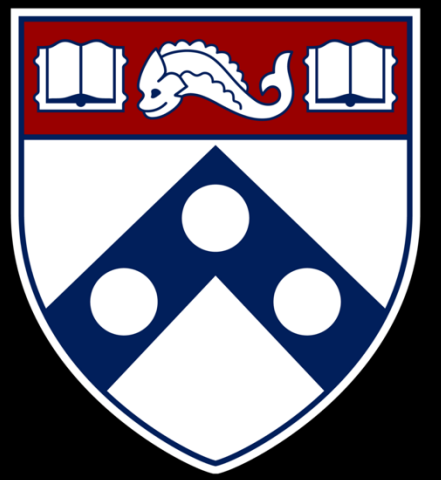
Secure Systems Engineering and Management

A Data-driven Approach



Measuring Secure Development Practices

Michael Hicks
UPenn CIS 7000-003
Spring 2026

Readings

(optional)

Build It, Break It, Fix It: Contesting Secure Development

Andrew Ruef Michael Hicks James Parker
Dave Levin Michelle L. Mazurek Piotr Mardziel†

University of Maryland †Carnegie Mellon University

ABSTRACT

Typical security contests focus on breaking or mitigating the impact of buggy systems. We present the Build-it, Break-it, Fix-it (BIBIFI) contest, which aims to assess the ability to securely build software, not just break it. In BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. During 2015, we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in other statically-typed languages were less likely to have a security flaw; build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

1. INTRODUCTION

Cybersecurity contests [24, 25, 11, 27, 13] are popular proving grounds for cybersecurity talent. Existing contests

experts have long advocated that achieving security in a computer system requires treating security as a first-order design goal [32], and is not something that can be added after the fact. As such, we should not assume that good breakers will necessarily be good builders [23], nor that top coders necessarily produce secure systems.

This paper presents **Build-it, Break-it, Fix-it** (BIBIFI), a new security contest with a focus on *building secure systems*. A BIBIFI contest has three phases. The first phase, *Build-it*, asks small development teams to build software according to a provided specification that includes security goals. The software is scored for being correct, efficient, and feature-ful. The second phase, *Break-it*, asks teams to find defects in other teams' build-it submissions. Reported defects, proved via test cases vetted by an oracle implementation, benefit a break-it team's score and penalize the build-it team's score; more points are assigned to security-relevant problems. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, asks builders to fix bugs and thereby get points back if the process discovers that distinct break-it test cases identify the same defect.

BIBIFI's design aims to minimize the manual effort of running a contest, helping it scale. BIBIFI's structure and scoring system also aim to encourage meaningful outcomes, e.g., to ensure that the top-scoring build-it teams really produce secure and efficient software. Behaviors that would thwart such outcomes are discouraged. For example, break-it teams may submit a limited number of bug reports per build-it submission, and will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to

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,jprider1,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,

developers [16, 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 [34]) 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, com-

Readings

Build It, Break It, Fix It: Contesting Secure Development

University of Maryland

Carnegie Mellon University

securely build software, not just break it. In BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. During 2015, we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in other statically-typed languages were less likely to have a security flaw; build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

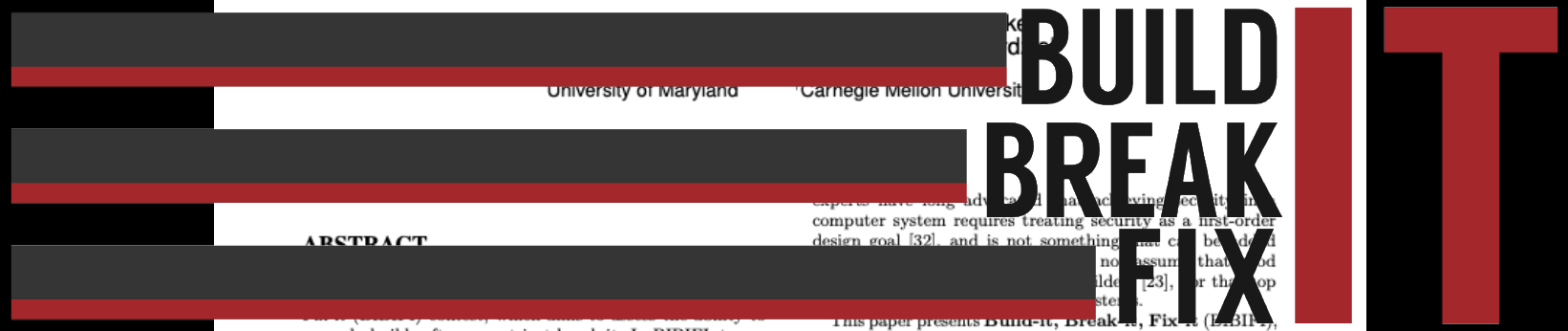
1. INTRODUCTION

Cybersecurity contests [24, 25, 11, 27, 13] are popular

computer system requires treating security as a first-order

This paper presents **Build-it, Break-it, Fix-it** (BIBIFI), a new security contest with a focus on *building secure systems*. A BIBIFI contest has three phases. The first phase, *Build-it*, asks small development teams to build software according to a provided specification that includes security goals. The software is scored for being correct, efficient, and feature-ful. The second phase, *Break-it*, asks teams to find defects in other teams' build-it submissions. Reported defects, proved via test cases vetted by an oracle implementation, benefit a break-it team's score and penalize the build-it team's score; more points are assigned to security-relevant problems. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, asks builders to fix bugs and thereby get points back if the process discovers that distinct break-it test cases identify the same defect.

BIBIFI's design aims to minimize the manual effort of running a contest, helping it scale. BIBIFI's structure and scoring system also aim to encourage meaningful outcomes, e.g., to ensure that the top-scoring build-it teams really produce secure and efficient software. Behaviors that would thwart such outcomes are discouraged. For example, break-it teams may submit a limited number of bug reports per build-it submission, and will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to



Overview

Round 1: **Build-it**
Teams build software to specification

2 weeks

Must satisfy basic **correctness** requirements; **optional features** and **good performance** for more points

Round 2: **Break-it**
Teams report bugs in submissions

2 weeks

Bug reports are (failing) executable **test cases**, including **exploits**

Round 3: **Fix-it**
Teams fix bugs found in their software

1 week

Doing so may **wipe out many bug reports in one go**: all count as the same bug

Last: Judges tally final results

Scoring System

- **Build-it Score**
  - **Gains** points for **good performance**
  - **Gains** points for implementing **optional features**
  - **Loses** points for *unique* **bugs found**
    - More points for (obviously) security-relevant bugs
- **Break-it Score**
  - **Gains** points for unique **bugs found**
    - Scaled by how many other teams found the same bug
- **Winners in both categories**

Build-it Round

- **Build software** according to the posted specification
  - Make it correct, feature-ful, efficient, and secure
    - The first three are assessed by (our) test cases (build-it round score)
    - The last is assessed by Break-it teams in round 2
- For many elements of the task, **teams may choose**
  - The software's internal **design** and **algorithms**
  - Which **optional features** to implement
  - What **programming language** to use
  - What development and testing **tools** to use
  - How to **divide tasks among team members**, etc.

Break-it Round

- Find **bugs and vulnerabilities** in submitted code
  - Provide an **exploit**, as defined by particular problem
- Teams will be given **access to the source code**
  - We provide scripts that teams can use to test projects against the standard tests, using a VM
- How teams go about this task is **up to them**, e.g.,
  - How to **divide up the task** among team members, and
  - whether (or how much) **testing** to use,
  - manual **code reviews**,
  - automated **dynamic/static analysis**, etc.

Fix-it Round

- Different teams may submit **different test cases** that identify the **same underlying bug**

```
void foo(char *str) {
  char buf[10];
  strcpy(buf,str);
}
```

- **`str` = “this is too long”**
- **`str` = “this is too long too”**
- **`str` = “and so is this string”**

- Build-it teams should only **lose points for each bug**, not for each test case that reveals it
- How to tell that test cases are “the same” ?

Fix-it Round

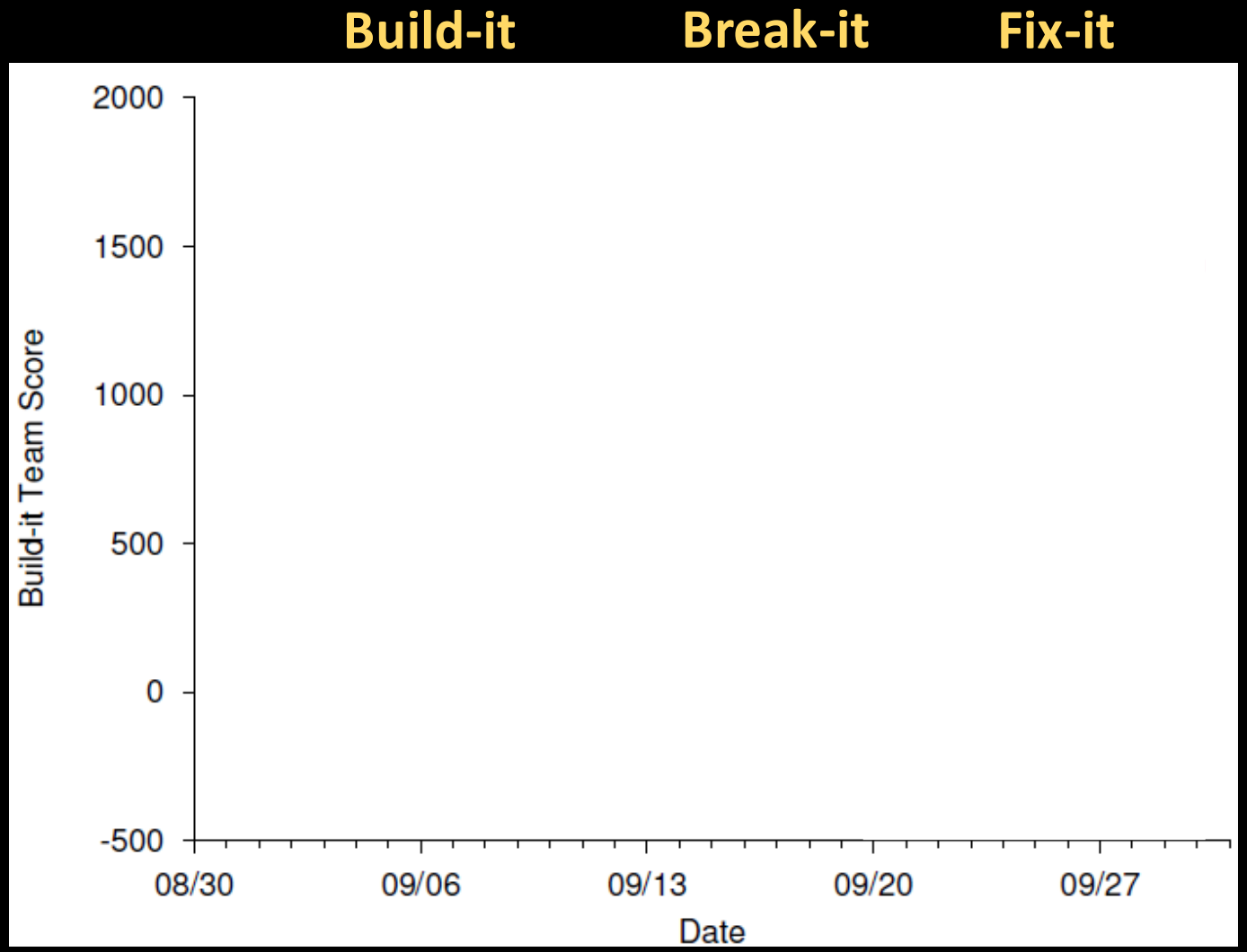
- Teams will receive the test cases during the fix-it round, and they can then **fix each bug identified**

```
void foo(char *str) {
  char buf[10];
  strlcpy(buf,str,10);
}
```

- `str` = “this is too long”
- `str` = “this is too long too”
- `str` = “and so is this string”

- **All test cases that pass** are unified to be the **same underlying bug**
  - Judges consider whether the fix is to a single bug
  - If not, the affected test cases will be scored individually

Builder score, as the contest progresses



Contest problems

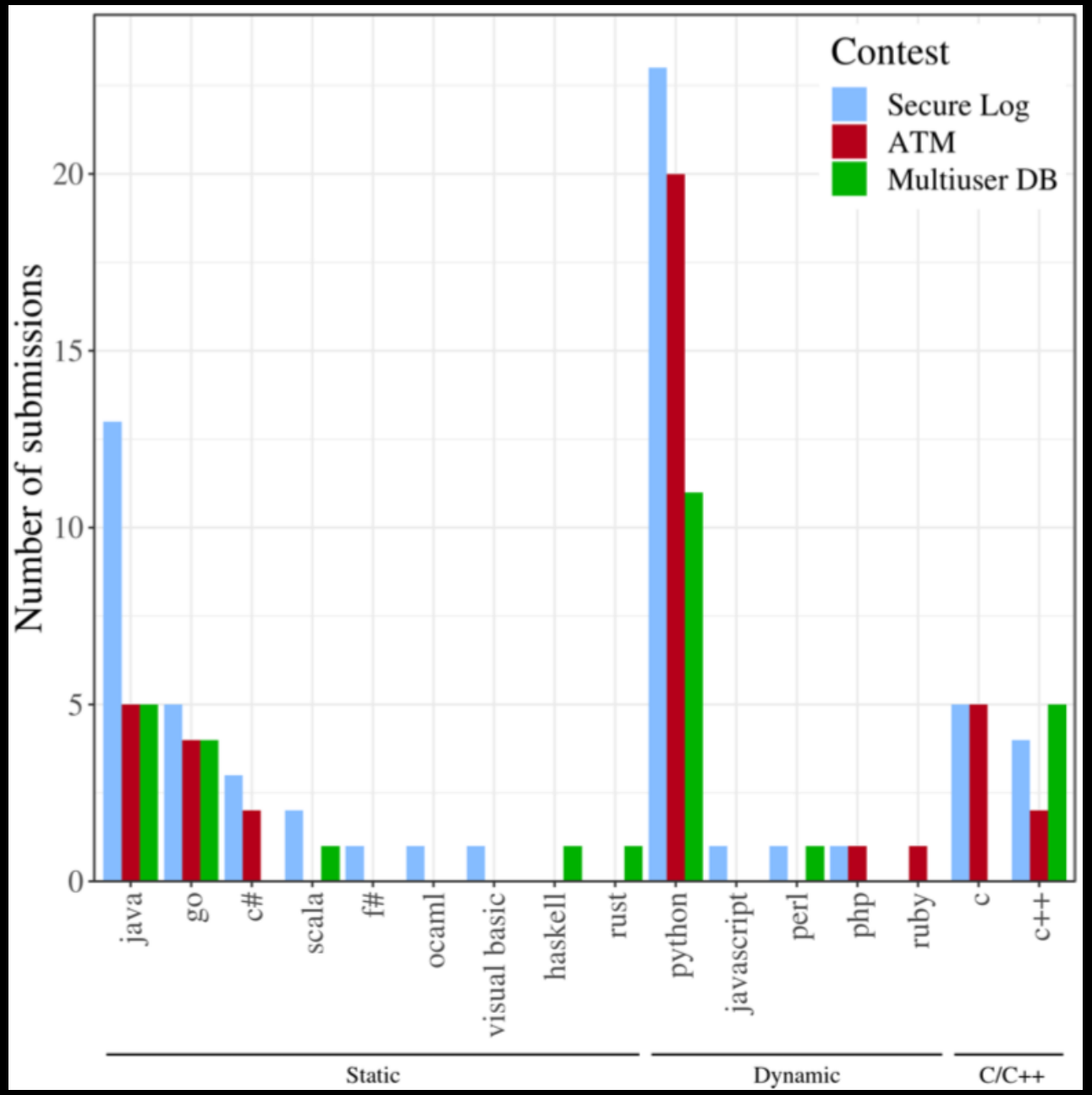
- Secure log of events at an art gallery
  - Commands to append records and query the log contents
  - Threat model:
    - Attacker has access to the log
    - Should be tamperproof and protect confidentiality
- Secure ATM
  - ATM communicates with Bank server to carry out transactions
  - Threat model: MITM can observe, send, drop messages, and simulate the ATM
- Multi-user DB [in extended version of paper]
  - Scriptable key-value store with RBAC policies with delegation
  - Threat model: Attacker as client; writes scripts to try to break RBAC implementation

Participant demographics

Contest	USA	India	Russia	Brazil	Other
Spring 2015	30	7	12	12	120
Fall 2015	64	14	12	20	110
Fall 2016	44	13	4	12	103

- Worldwide participant pool (mostly non-US)
  - 156 teams, 406 people
- Average 9 years programming experience
  - 1/3 have CS degree
- Most participants part of MOOC
  - Four courses of security training
- Average team size: ~2

Contest	Spring 15	Fall 15	Fall 16
Problem	Secure Log	ATM	Multiuser DB
# Contestants	156	145	105
% Male	91 %	91 %	84 %
% Female	5 %	8 %	4 %
Age (mean/min/max)	34.8/20/61	32.2/17/69	29.9/18/55
% with CS degrees	35 %	35 %	39 %
Years programming	9.6/0/30	9.4/0/37	9.0/0/36
# Build-it teams	61	40	29
Build-it team size	2.2/1/5	3.1/1/6	2.5/1/8
# Break-it teams (that also built)	65 (58)	43 (35)	33 (22)
Break-it team size	2.4/1/5	3.1/1/6	2.6/1/8
# PLs known per team	6.8/1/22	9.1/1/20	7.8/1/17
% MOOC	100 %	84 %	65 %



Submission features

- **Count** by language
- Grouped by **category**
  - **Static**ally type-safe (49)
  - **Dynamic**ally typed (60)
    - 54 are Python!
  - **C/C++** (21)

The Four Core Analyses

#	Outcome Variable	Model Type	Table
1	Ship score (continuous)	Linear regression	Table 5
2	Security bug found? (binary)	Logistic regression	Table 7
3	Break-it score (continuous)	Linear regression	Table 10
4	Security bug count (continuous)	Linear regression	Table 11

All four use the same general approach: select candidate factors, test all subsets, pick the best model via AIC

Ship Score Model (Table 5)

Outcome: Build-it team's ship score (continuous, points-based)

C/C++ teams scored ~113–133 points higher than other language categories

Factor	Coef.	CI	*p*-value
Secure Log	—	—	—
ATM	-47.708	[-110.34, 14.92]	0.138
Multiuser DB	**-163.901**	**[-234.2, -93.6]**	**<0.001***
C/C++	—	—	—
Statically typed	**-112.912**	**[-192.07, -33.75]**	**0.006***
Dynamically typed	**-133.057**	**[-215.26, -50.86]**	**0.002***
# Languages known	6.272	[-0.06, 12.6]	0.054
Lines of code	-0.023	[-0.05, 0.01]	0.118

$R^2 = 0.232$

Security Bug Model (Table 7)

Outcome: Was a security bug found in this team's submission?

So: **logistic regression**

ATM contest: vastly more security bugs than Secure Log

C/C++ submissions were **~11× more likely** to have a security bug (1/0.089 ≈ 11.2)

Factor	Coef.	Exp(coef)	Exp CI	*p*-value
Secure Log	—	—	—	—
ATM	**4.639**	**103.415**	**[18, 594.11]**	**<0.001***
Multiuser DB	**3.462**	**31.892**	**[7.06, 144.07]**	**<0.001***
C/C++	—	—	—	—
Statically typed	**-2.422**	**0.089**	**[0.02, 0.51]**	**0.006***
Dynamically typed	-0.99	0.372	[0.07, 2.12]	0.266
# Team members	-0.35	0.705	[0.5, 1]	0.051
Knowledge of C	-1.44	0.237	[0.05, 1.09]	0.064
Lines of code	0.001	1.001	[1, 1]	0.090

Nagelkerke $R^2 = 0.619$

Odds Ratios and Exponential Coefficients

Recall:

- **Odds ratios (e^{β})** — The paper reports Exp(coef) and Exp CI columns: exponentiated coefficients and their confidence intervals
- **Interpreting direction** — Exp(coef) < 1 means *lower* likelihood
- **Confidence intervals on odds ratios** — Statically typed: [0.02, 0.51], entirely below 1, confirming the protective effect

Recall our pitfall: “Higher CVSS increases exploitation by 0.52” ❌

The paper correctly says “11× more likely” and not the raw coefficient

Break-it Models (Tables 10 and 11)

Two more linear regressions, now for the breaking phase

Model	Key Significant Factors
Break-it score (Table 10)	More team members (+387 pts each, p = 0.028); ATM teams scored lower
Security bug count (Table 11)	More team members (+1.2 bugs each, p = 0.006); Build participants found +4 more bugs (p = 0.045)

$R^2 = 0.15$ and $R^2 = 0.203$ respectively

Factors That Didn’t Make the Cut

Notably absent from final models:

- **Advanced techniques** (fuzzing, static analysis) — dropped during model selection, not significant
- **MOOC participation** — security education didn’t significantly help

But: Non-significant results are still informative! The paper discusses these:

> “Making use of advanced analysis techniques did not factor into the final model… such techniques tend to find generic errors such as crashes, bounds violations… Security violations for our problems are more often semantic”

Model Selection via AIC (new)

What the paper does: “We test models with all possible combinations of our chosen potential factors and select the model with the minimum Akaike Information Criterion (AIC)”

- AIC is an *information-theoretic* criterion: AIC = 2k − 2 ln(L), where k = number of parameters and L = likelihood
  - Lower AIC = better balance of fit and parsimony
- In lecture, we used likelihood-ratio (LR) tests to compare nested models (reduced vs. full)

Key difference: LR tests compare two *specific* nested models; AIC can be used to rank *any set* of models, including non-nested ones

The Iterative Model-Building Process

1. Select candidate factors based on domain knowledge (Tabs 4 and 9)
2. Limit the number of factors based on **power analysis**
3. Test all possible combinations of factors
4. Select the model with minimum AIC
5. Report the final model

What Is Power Analysis?

Problem: How many factors can we include in our model without overfitting?

Statistical power = the probability of detecting a real effect if one exists (i.e., correctly rejecting H_0)

- Convention: aim for power ≥ 0.75 or 0.80
- Recall: failing to reject H_0 when it's false = **Type II error**
- Power = 1 – P(Type II error)

Power depends on three things:

Factor	Effect on Power
Sample size (N)	Larger N → more power
Effect size (f^2)	Larger effect → easier to detect
Number of parameters (k)	More parameters → less power per parameter

How the Paper Uses Power Analysis

Power analysis in the paper:

1. Given: N = 130 build-it teams
2. Goal: “medium” effect size or Cohen’s $f^2 = 0.15$ (equivalent to $R^2 \approx 0.13$)
3. Goal: power = 0.75,

Result: We are limited to **10 degrees of freedom** (i.e., k = ~10 parameters)

So we pre-commit to a small set of factors (Table 4) before looking at results — this prevents overfitting and data mining (“p hacking”)

Contrast with a naïve approach:

- ❌ Throw 30 variables into a regression with 130 observations → spurious results
- ✅ Power analysis says: with N = 130, you can responsibly test ~10 factors for medium effects

A Different Effect Size: Cohen's f^2

From our lectures: Cohen's d for comparing two group means

In the paper: Cohen's f^2 for regression effect sizes

"Our modeling was designed for a prospective effect size roughly equivalent to Cohen's medium effect heuristic, $f^2 = 0.15$"

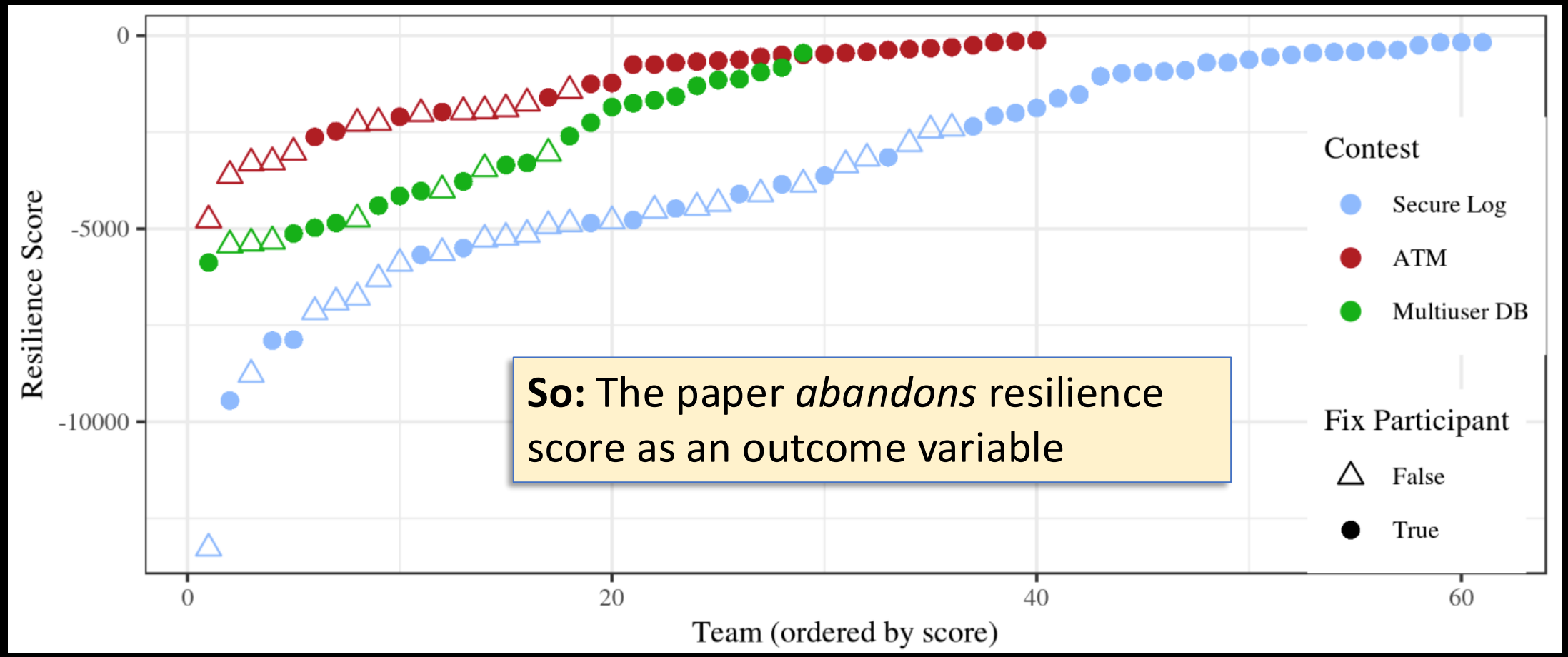
This corresponds to $R^2 = 0.13$ — the model needs to explain at least 13% of variance to be detectable

Measure	Context	"Medium" threshold
Cohen's d	Two-group comparison	0.5
Cohen's f^2	Regression models	0.15

Pitfalls avoided

- Correlation ≠ Causation
  - “This was not a completely controlled experiment (e.g., we do not use random assignment), so our models demonstrate correlation rather than causation.”
- Independent samples
  - Build it and Break it participants overlap (Ok – not combined in analysis)
  - Overlap in participants between contests? Added ”Contest” as factor
- Honest about limitations
  - Resilience score issue, limitations of self reported data

Survivor Bias in Resilience Scores



Pitfall: Multiple Models, No Correction

From our lecture: Running many tests inflates false positive rates

The paper fits four separate regression models, each testing multiple factors

- No explicit correction for multiple comparisons across models
- The all-subsets AIC approach tests many model specifications within each analysis

Mitigating factors:

- AIC inherently penalizes model complexity (acts as a soft correction)
- The factors were pre-specified based on domain knowledge, not data-mined
- Power analysis limits the number of factors tested

But: with enough model combinations, some “significant” factors may be false positives

Mapping to Our Lectures

Lecture Topic	Used in Paper?	Where
Linear regression	✅	Tables 5, 10, 11
Logistic regression	✅	Table 7
Dummy coding / reference levels	✅	Language category, Contest
Odds ratios (e^β)	✅	Table 7
Confidence intervals	✅	All tables
R^2 / model fit	✅	All models
Effect sizes (Cohen's d)	⚠ Variant (f^2)	Power analysis
LR test (model comparison)	⚠ AIC instead	Model selection
Pseudo-R^2	⚠ Nagelkerke, not	Table 7

Readings

(optional)

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,jprider1,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, developers [16, 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 [34]) 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.

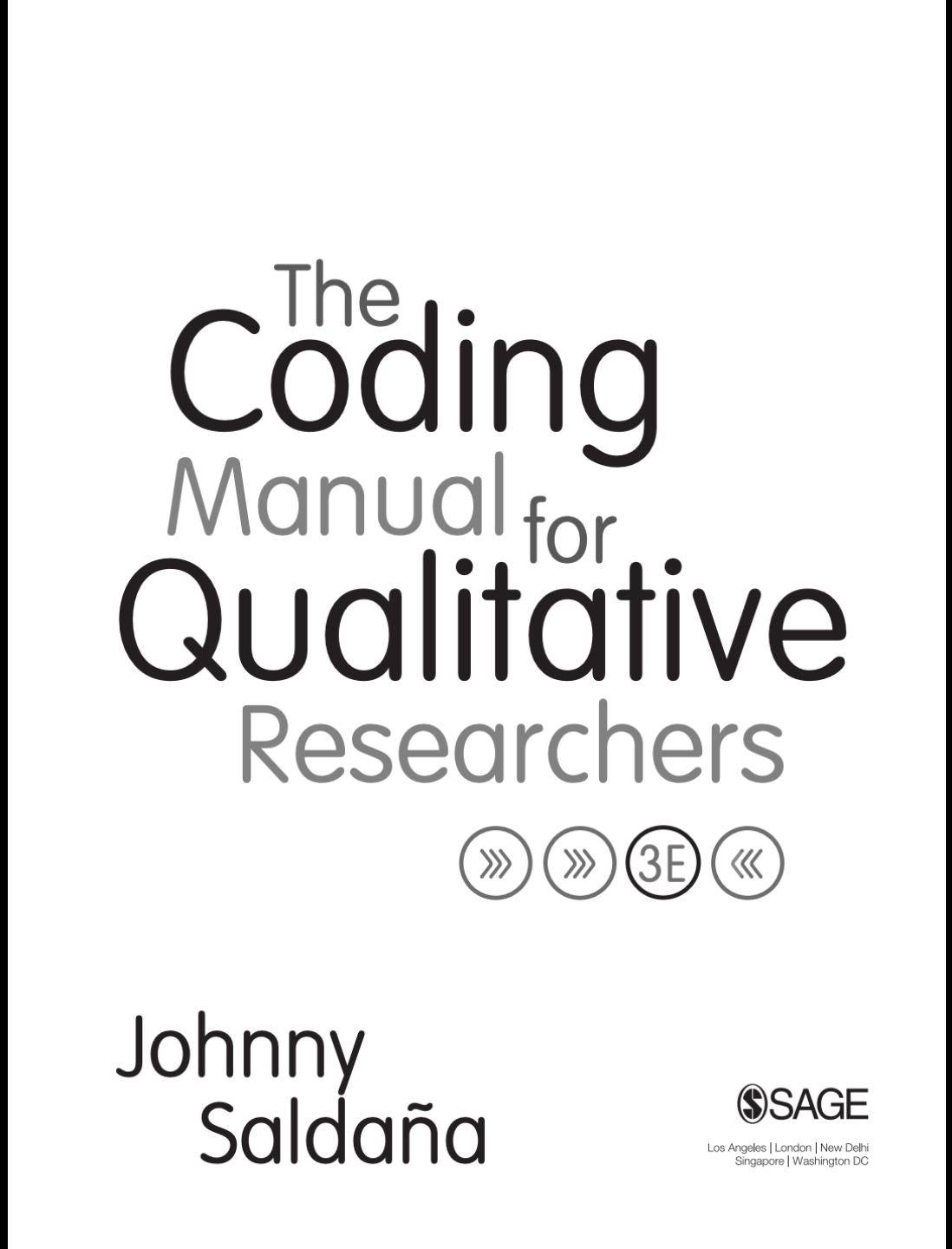
The BIBIFI competition provides a unique and valuable vantage point for examining the vulnerability landscape, com-

Approach

- Examined each project and vulnerability in detail
  - 94 projects
  - Breaker-identified (866 submitted exploits) and researcher-identified (manual analysis)
  - In total, 182 distinct vulnerabilities
- Iterative open and axial coding
- Qual and quant analysis on resulting categories

What Is Qualitative Coding?

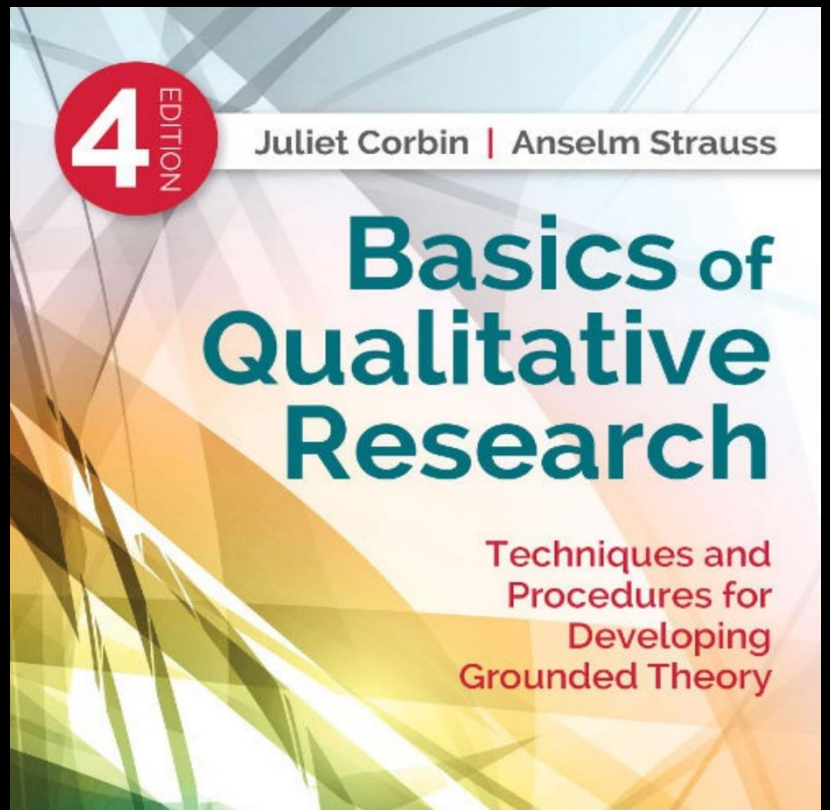
- A **code** is a word or short phrase that captures the meaning of a piece of data — it is "primarily an interpretive act" (Saldaña, Ch. 1)
- Coding is **cyclical**: first cycle codes emerge from the data, then get refined, reorganized, and consolidated into categories and themes through second cycle coding
- The BIBIFI paper used **iterative open coding** (codes emerge from data, not predetermined) and **axial coding** (grouping codes into higher-level types)



Further reading:

Saldaña, *The Coding Manual for Qualitative Researchers*, **Ch. 1**;

Strauss & Corbin, *Basics of Qualitative Research* (1990)



How BIBIFI Applied It

1. Two researchers **cooperatively examined 11 projects** (42 vulnerabilities) to build an initial codebook
2. Then coded independently in **rounds of ~30 breaks**, meeting after each round to discuss disagreements and refine the codebook
3. Process continued for **~6 months** until inter-rater reliability exceeded $\alpha > 0.80$ (Krippendorff's α) on all variables
4. Remaining 34 projects divided and coded separately

Result: 182 unique vulnerabilities coded across four variables (Type, Attacker Control, Discovery Difficulty, Exploit Difficulty)

The Codebook

Each vulnerability was labeled on four variables with defined levels:

Variable	Levels	Description	Alpha [38]
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

Axial coding then grouped the 23 specific issues into three high-level types:

- **No Implementation** — didn't attempt a necessary security mechanism
- **Misunderstanding** — attempted it but made a conceptual error
- **Mistake** — had the right idea but made a programming slip

Why This Approach?

- Qualitative coding lets researchers **systematically characterize** unstructured data (source code, exploits) in a reproducible way
- The codebook with inter-rater reliability ensures findings aren't just one person's opinion
- Coded categories then become variables for **quantitative analysis** (Chi-squared tests, Poisson regression) — the two approaches are complementary

Saldaña (Ch. 1): "Quantitative analysis calculates the mean. Qualitative analysis calculates meaning."

Vulnerability classes

No Implementation

Misunderstanding

Mistake

Intuitive

Unintuitive

Bad Choice

Conceptual Error

Vulnerability classes

No Implementation

Intuitive

Missed something “Intuitive”

- No encryption
- No access control

Vulnerability classes

No Implementation

Unintuitive

Missed something “Unintuitive”

- No MAC
- Side channel leakage
- No replay prevention

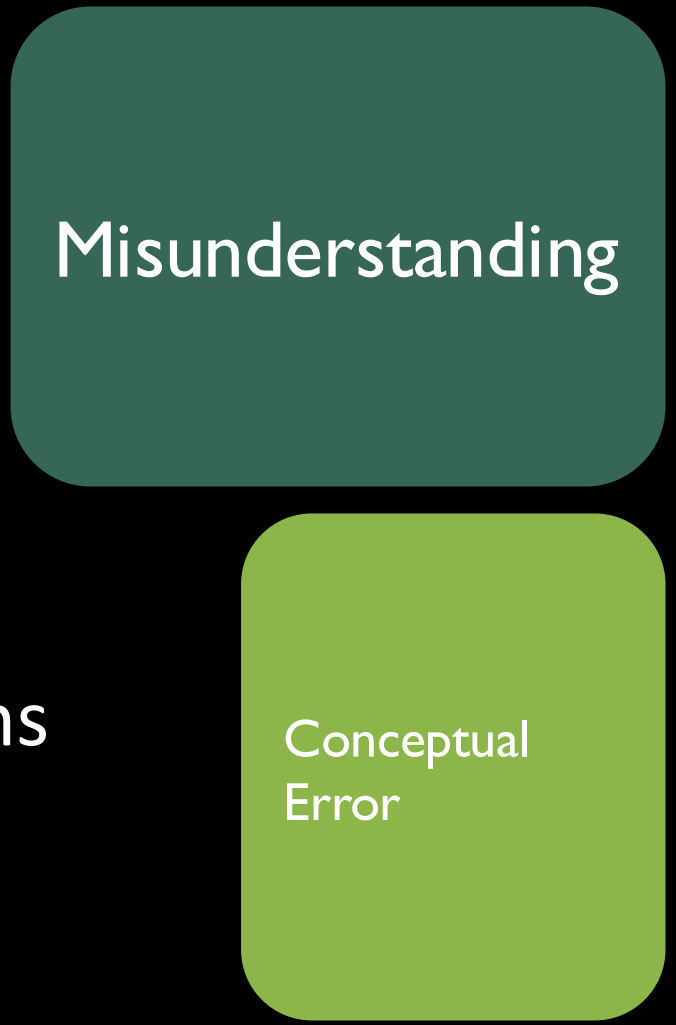
45% of projects

Vulnerability classes

Made a “bad choice”

- Weak algorithms
- Homemade encryption
- strcpy()

Vulnerability classes



Made a “conceptual error”

- Insufficient randomness
- Disabling default protections

44% of projects

Vulnerability classes

Made a programming “mistake”

- Control flow error
- Skipped algorithmic step

21% of projects

Summary of data analysis

- No implementation & misunderstanding more common (78%) than mistake (21%)
  - Mistake: control error, skipped step
- Unintuitive requirements missed or implemented incorrectly much more often (45%) than intuitive ones
  - Unintuitive: MAC; avoiding side channels and/or replays
  - Intuitive: Encryption for privacy; access control
- Implementation complexity breeds mistakes
  - Failure to localize functionality, minimize TCB, completely mediate
- Mistakes readily exploited
  - Almost always result in contestant attacks

Recommendations

- Simplify API design
  - Build in security primitives and focus on common use-cases
- Indicate security impact of non-default use in API documentation
  - Explain the negative effects of turning off certain things
- Expand capabilities of vulnerability analysis tools
  - More emphasis on design-level conceptual issues