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

JAMES PARKER, MICHAEL HICKS, ANDREW RUEF, MICHELLE L. MAZUREK,
DAVE LEVIN, and DANIEL VOTIPKA, University of Maryland, USA
PIOTR MARDZIEL, Carnegie Mellon University Silicon Valley, USA
KELSEY R. FULTON, University of Maryland, USA

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, and so on, that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. We ran three contests involving a total of 156 teams and three different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in a statically type safe language were 11× less likely to have a security flaw than C/C++ submissions. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

CCS Concepts: • **Security and privacy** → *Software security engineering*; • **Social and professional topics** → *Computing education*; • **Software and its engineering** → *Software development techniques*;

Additional Key Words and Phrases: Contest, security, education, software, engineering

ACM Reference format:
James Parker, Michael Hicks, Andrew Ruef, Michelle L. Mazurek, Dave Levin, Daniel Votipka, Piotr Mardziel, and Kelsey R. Fulton. 2020. Build It, Break It, Fix It: Contesting Secure Development. *ACM Trans. Priv. Secur.* 23, 2, Article 10 (April 2020), 36 pages.
https://doi.org/10.1145/3383773

1 INTRODUCTION

Cybersecurity contests [3, 8, 30, 46, 53] are popular proving grounds for cybersecurity talent. Existing contests largely focus on *breaking* (e.g., exploiting vulnerabilities or misconfigurations) and

This project was supported with gifts from Accenture, AT&T, Booz Allen Hamilton, Galois, Leidos, Patriot Technologies, NCC Group, Trail of Bits, Synposis, ASTech Consulting, Cigital, SuprTek, Cyberpoint, and Lockheed Martin; by a 2016 Google Faculty Research Award; by grants from the NSF under Awards No. EDU-1319147 and No. CNS-1801545; by DARPA under Contract No. FA8750-15-2-0104; and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement No. 70NANB15H330.
Authors' addresses: J. Parker, M. Hicks, A. Ruef, M. L. Mazurek, D. Levin, D. Votipka, and K. R. Fulton, University of Maryland, 8125 Paint Branch Dr., College Park, MD, 20740, USA; emails: jp@jamesparker.me, {mwh, awruef, mmazurek, dml, dvotipka, kfulton}@cs.umd.edu; P. Mardziel, Carnegie Mellon University Silicon Valley, NASA Research Park, Moffett Field, CA, 94035, USA; email: piotrm@gmail.com.

2471-2566/2020/04-ART10 $15.00
https://doi.org/10.1145/3383773

mitigation (e.g., rapid patching or reconfiguration). They do not, however, test contestants' ability to *build* (i.e., design and implement) systems that are secure in the first place. Typical programming contests [1, 7, 9] do focus on design and implementation, but generally ignore security. This state of affairs is unfortunate, because experts have long advocated that achieving security in a computer system requires treating security as a first-order design goal [58] 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 [41] or that top coders can produce secure systems.

This article presents **Build-it, Break-it, Fix-it** (BIBIFI), a new security contest focused 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 including security goals. The software is scored for being correct, efficient, and featureful. 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 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 top-scoring build-it teams really produce secure and efficient software. 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 look for bugs broadly (in many submissions) and deeply (to uncover hard-to-find bugs).

In addition to providing a novel educational experience, BIBIFI presents an opportunity to study the building and breaking process scientifically. In particular, BIBIFI contests may serve as a quasi-controlled experiment that correlates participation data with final outcome. By examining artifacts and participant surveys, we can study how the choice of build-it programming language, team size and experience, code size, testing technique, and so on, are associated with a team's (non)success in the build-it or break-it phases. To the extent that contest problems are realistic and contest participants represent the professional developer community, the results of this study may provide useful empirical evidence for practices that help or harm real-world security. Indeed, the contest environment could be used to incubate ideas to improve development security, with the best ideas making their way to practice.

This article studies the outcomes of three BIBIFI contests that we held during 2015 and 2016, involving three different programming problems. The first contest asked participants to build a *secure, append-only log* for adding and querying events generated by a hypothetical art gallery security system. Attackers with direct access to the log, but lacking an "authentication token," should not be able to steal or corrupt the data it contains. The second contest asked participants to build a pair of *secure, communicating programs*, one representing an ATM and the other representing a bank. Attackers acting as a man in the middle (MITM) should neither be able to steal information (e.g., bank account names or balances) nor corrupt it (e.g., stealing from or adding money to accounts).[1] The third contest required participants to build a *access-controlled, multiuser data server* that protects the data of users. Users are authenticated via password checking, and access control with delegation restricts how data is read and modified. All contests drew participants from a MOOC (Massive Online Open Courseware) course on cybersecurity. MOOC participants had an average of 10 years of programming experience and had just completed a four-course

[1]Such attacks are realistic in practice, as detailed in a 2018 analysis of actual ATMs [63].

sequence including courses on secure software and cryptography. The second and third contests also involved graduate and undergraduate students with less experience and training. The first contest had 156 MOOC participants (comprising 68 teams). The second contest was composed of 122 MOOC participants (comprising 41 teams) and 23 student participants (comprising 7 teams). The last contest had 68 MOOC participants (comprising 25 teams) and 37 student participants (comprising 15 teams).

BIBIFI's design permitted it to scale reasonably well. For example, one full-time person and two part-time judges ran the first contest in its entirety. This contest involved over one hundred participants who submitted more than 20,000 test cases. And yet, organizer effort was limited to judging that the few hundred submitted fixes addressed only a single conceptual defect; other work was handled automatically or by the participants themselves.

Rigorous quantitative analysis of the contests revealed several interesting, statistically significant effects. Considering build-it scores: Writing code in C/C++ increased build-it scores initially but also increased chances of a security bug being found later; C/C++ programs were 11× more likely to have a reported security bug than programs written in a statically type-safe language. Considering break-it scores: Larger teams found more bugs during the break-it phase, and teams that also qualified during the build-it phase found more security bugs than those that did not. Other trends emerged, but did not reach significance. Build-it teams with more developers tended to produce more secure implementations. More programming experience was helpful for breakers.

We manually examined both build-it and break-it artifacts. Successful build-it teams typically employed third-party libraries—e.g., SSL, NaCL, and BouncyCastle—to implement cryptographic operations and/or communications, which provided primitives such as randomness, nonces, and so on. Unsuccessful teams typically failed to employ cryptography, implemented it incorrectly, used insufficient randomness, or failed to use authentication. Break-it teams found clever ways to exploit security problems; some MITM implementations were quite sophisticated.

This article extends a previously published conference paper [56] to include data from an additional contest run, as well as more details about our overall experience. Since publishing that paper, we have made the BIBIFI code and infrastructure publicly available so that others may run their own competitions. We, and those at other universities, including Universität Paderborn, University of Pennsylvania, Texas A&M University, and Carnegie Mellon University, have already used the BIBIFI infrastructure to run contests in a classroom setting. More information, data, and opportunities to participate are available at https://builditbreakit.org and the BIBIFI codebase is at https://github.com/plum-umd/bibifi-code.

In summary, this article makes two main contributions. First, it presents BIBIFI, a security contest that encourages building, not just breaking. Second, it presents a detailed description of three BIBIFI contests along with success and failure stories as well as a quantitative analysis of the results. The article is organized as follows. We present the design of BIBIFI in Section 2 and describe specifics of the contests we ran in Section 3. We present success and failure stories of contestants in Section 4 and a quantitative analysis of the data we collected in Section 5. We review related work in Section 6 and conclude in Section 7.

2 BUILD-IT, BREAK-IT, FIX-IT

This section describes the goals, design, and implementation of the BIBIFI competition. At the highest level, our aim is to create an environment that closely reflects real-world development goals and constraints, and to encourage build-it teams to write the most secure code they can, and break-it teams to perform the most thorough, creative analysis of others' code they can. We achieve this through a careful design of how the competition is run and how various acts are scored

(or penalized). We also aim to minimize the manual work required of the organizers—to allow the contest to scale—by employing automation and proper participant incentives.

2.1 Competition Phases

We begin by describing the high-level mechanics of a BIBIFI competition. BIBIFI may be administered online, rather than on-site, so teams may be geographically distributed. The contest comprises three phases, each of which lasts about two weeks for the contests we describe in this article.

BIBIFI begins with the **build-it phase**. Registered contestants aim to implement the target software system according to a published specification created by the contest organizers. A suitable target is one that can be completed by good programmers in a short time (just about two weeks, for the contests we ran), is easily benchmarked for performance, and has an interesting attack surface. The software should have specific security goals—e.g., protecting private information or communications—which could be compromised by poor design and/or implementation. The software should also not be too similar to existing software to ensure that contestants do the coding themselves (while still taking advantage of high-quality libraries and frameworks to the extent possible). The software must build and run on a standard Linux VM made available prior to the start of the contest. Teams must develop using Git [5]; with each push, the contest infrastructure downloads the submission, builds it, tests it (for correctness and performance), and updates the scoreboard. Section 3 describes the three target problems we developed: (1) an append-only log (aka, Secure Log), (2) a pair of communicating programs that simulate a bank and ATM (aka, ATM), and (3) a multi-user data server with custom access control policies (aka, Multiuser DB).

The next phase is the **break-it phase**. Break-it teams can download, build, and inspect all qualifying build-it submissions, including source code; to qualify, the submission must build properly, pass all correctness tests, and not be purposely obfuscated (accusations of obfuscation are manually judged by the contest organizers). We randomize each break-it team's view of the build-it teams' submissions, but organize them by meta-data, such as programming language used. (Randomization aims to encourage equal scrutiny of submissions by discouraging break-it teams from investigating projects in the same order.) When they think they have found a defect, breakers submit a test case that exposes the defect and an explanation of the issue. We impose an upper bound on the number of test cases a break-it team can submit against a single build-it submission, to encourage teams to look at many submissions. BIBIFI's infrastructure automatically judges whether a submitted test case truly reveals a defect. For example, for a correctness bug, it will run the test against a reference implementation ("the oracle") and the targeted submission, and only if the test passes on the former but fails on the latter will it be accepted. Teams can also earn points by reporting bugs in the oracle, i.e., where its behavior contradicts the written specification; these reports are judged by the organizers. More points are awarded to clear security problems, which may be demonstrated using alternative test formats. The auto-judgment approaches we developed for the three different contest problems are described in Section 3.

The final phase is the **fix-it phase**. Build-it teams are provided with the bug reports and test cases implicating their submission. They may fix flaws these test cases identify; if a single fix corrects more than one failing test case, the test cases are "morally the same," and thus points are only deducted for one of them. The organizers determine, based on information provided by the build-it teams and other assessment, whether a submitted fix is "atomic" in the sense that it corrects only one conceptual flaw; if not, the fix is rejected.

Once the final phase concludes, prizes are awarded to the builders and breakers with the best scores, as determined by the scoring system described next.

2.2 Competition Scoring

BIBIFI's scoring system aims to encourage the contest's basic goals, which are that the winners of the build-it phase truly produced the highest quality software, and that the winners of the break-it phase performed the most thorough, effective analysis of others' code. The scoring rules, and the fact that scores are published in real time while the contest takes place, create incentives for good behavior (and disincentives for bad behavior).

2.2.1 Build-it Scores. To reflect real-world development concerns, the winning build-it team would ideally develop software that is correct, secure, featureful, and efficient. While security is of primary interest to our contest, developers in practice must balance other aspects of quality against security [12, 66], creating trade-offs that cannot be ignored if we wish to motivate realistic developer decision-making.

As such, each build-it team's score is the sum of the *ship* score[2] and the *resilience* score. The ship score is composed of points gained for correctness tests and performance tests. Each mandatory correctness test is worth M points, for some constant M, while each optional correctness test is worth $M/2$ points. Each performance test has a numeric measure depending on the specific nature of the programming project—e.g., latency, space consumed, files left unprocessed—where lower measures are better. A test's worth is $M \cdot (\mathit{worst} - v)/(\mathit{worst} - \mathit{best})$, where v is the measured result, *best* is the measure for the best-performing submission, and *worst* is the worst performing. As such, each performance test's value ranges from 0 to M. As scores are published in real time, teams can see whether they are scoring better than other participants. Their relative standing may motivate them to improve their implementation to improve its score before the build-it phase ends.

The resilience score is determined after the break-it and fix-it phases, at which point the set of unique defects against a submission is known. For each *unique* bug found against a team's submission we subtract P points from its resilience score; as such, the best possible resilience score is 0. For correctness bugs, we set P to $M/2$; for crashes that violate memory safety, we set P to M; and for exploits and other security property failures, we set P to $2M$. (We discuss the rationale for these choices below.) Once again, real-time scoring helps incentivize fixing, to get points back.

2.2.2 Break-it Scores. Our primary goal with break-it teams is to encourage them to find as many defects as possible in the submitted software, as this would give greater confidence in our assessment that one build-it team's software is of higher quality than another's. While we are particularly interested in obvious security defects, correctness defects are also important, as they can have non-obvious security implications.

A break-it team's score is the summed value of all defects they have found, using the above P valuations. This score is shown in real time during the break-it phase, incentivizing teams to improve their standing. After the fix-it phase, this score is reduced. In particular, if a break-it team submitted multiple test cases against a project that identify the same defect, the duplicates are discounted. Moreover, each of the N break-it teams' scores that identified the same defect are adjusted to receive P/N points for that defect, splitting the P points among them.

Through a combination of requiring concrete test cases and scoring, BIBIFI encourages break-it teams to follow the spirit of the competition. First, by requiring them to provide test cases as evidence of a defect or vulnerability, we ensure they are providing useful bug reports. By providing 4× more points for security-relevant bugs than for correctness bugs, we nudge break-it teams to look for these sorts of flaws and to not just focus on correctness issues. (But a different ratio might work better; see below.) Because break-it teams are limited to a fixed number of test cases per submission, they are discouraged from submitting many tests they suspect are "morally the

[2]The name is meant to evoke a quality measure at the time software is shipped.

Table 1. Example Scoring Results for a Three-team Contest with One Optional Test and One Performance Test ($M = 50$)

Team	Correctness Tests	Optional Test	Performance Test Runtime (s)	Ship Score	Bugs Against	Resilience Score	Bugs Reported	Break Score
Team 1	Pass	Fail	4	300	S_1	200	S_2	100
Team 2	Pass	Pass	5	315	C_1	290	S_1	50
Team 3	Pass	Fail	9	250	S_2	150	S_1, C_1	75

Bugs S_1 and S_2 are security vulnerabilities; bug C is a correctness bug.

same"; as they could lose points for them during the fix-it phase, they are better off submitting tests demonstrating different bugs. Limiting per-submission test cases also encourages examining many submissions. Finally, because points for defects found by other teams are shared, break-it teams are encouraged to look for hard-to-find bugs, rather than just low-hanging fruit.

Scoring example. Table 1 presents an example scoreboard for a three-team contest. For simplicity, there is only one optional test and one performance test. We set M to be 50 points.

Consider the *ship score*. All teams receive 250 points for passing all correctness tests. Team 2 receives 25 additional points for implementing the optional test. Team 1 receives 50 additional points for having the fastest performance test; team 2 gets 40 points for being relatively 20% slower; and team 3 receives no performance points, since their implementation is slowest. Now consider *resilience score*. This is a team's ship score minus points lost for each unique bug against its implementation. Each team has one bug against it, where bugs S_1 and S_2 are security vulnerabilities (100 points), while bug C_1 is a correctness bug (25 points). Finally, a team's *break score* is the number of points it receives for discovering bugs, split between teams that discover the same bug. Teams 2 and 3 split the 100 points for discovering exploit S_1 against team 1.

2.2.3 Discouraging Collusion. BIBIFI contestants may form teams however they wish, and they may participate remotely. This encourages wider participation, but it also opens the possibility of collusion between teams, as there cannot be a judge overseeing their communication and coordination. There are three broad possibilities for collusion, each of which BIBIFI's scoring discourages.

First, two break-it teams could consider sharing bugs they find with one another. By scaling the points each finder of a particular bug obtains, we remove incentive for them to both submit the same bugs, as they would risk diluting how many points they both obtain.

The second class of collusion is between a build-it team and a break-it team, but neither have incentive to assist one another. The zero-sum nature of the scoring between breakers and builders places them at odds with one another; revealing a bug to a break-it team hurts the builder, and not reporting a bug hurts the breaker.

Finally, two build-it teams could collude, for instance by sharing code with one another. It might be in their interests to do this in the event that the competition offers prizes to two or more build-it teams, since collusion could obtain more than one prize-position. We use judging and automated tools (and feedback from break-it teams) to detect if two teams share the same code (and disqualify them), but it is not clear how to detect whether two teams provided out-of-band feedback to one another prior to submitting code (e.g., by holding their own informal "break-it" and "fix-it" stages). We view this as a minor threat to validity; at the surface, such assistance appears unfair, but it is not clear that it is contrary to the goals of the contest, that is, to developing secure code.

2.3 Discussion

The contest's design also aims to enable scalability by reducing work on contest organizers. In our experience, BIBIFI's design succeeds at what it sets out to achieve, but has limitations.

Minimizing manual effort. Once the contest begins, manual effort by the organizers is limited by design. All bug reports submitted during the break-it phase are automatically judged by the oracle; organizers only need to vet any bug reports against the oracle itself. Organizers may also need to judge accusations by breakers of code obfuscation by builders. Finally, organizers must judge whether submitted fixes address a single defect; this is the most time consuming task. It is necessary, because we cannot automatically determine whether multiple bug reports against one team map to the same software defect; techniques for automatic testcase deduplication are still a matter of research (see Section 6). As such, we incentivize build-it teams to demonstrate overlap through fixes, which organizers manually confirm address only a single defect, not several.

Previewing some of the results presented later, we can confirm that the design works reasonably well. For example, as detailed in Table 6, 68 teams submitted 24,796 test cases for the Secure Log contest. The oracle auto-rejected 15,314 of these, and build-it teams addressed 2,252 of those remaining with 375 fixes, a 6× reduction. Most confirmations that a fix addresses a single bug took 1-2 minutes. Only 30 of these fixes were rejected. No accusations of code obfuscation were made by break-it teams, and few bug reports were submitted against the oracle. All told, the Secure Log contest was successfully managed by one full-time person, with two others helping with judging.

Limitations. While we believe BIBIFI's structural and scoring incentives are properly designed, we should emphasize several limitations.

First, there is no guarantee that all implementation defects will be found. Break-it teams may lack the time or skill to find problems in all submissions, and not all submissions may receive equal scrutiny. Break-it teams may also act contrary to incentives and focus on easy-to-find and/or duplicated bugs, rather than the harder and/or unique ones. In addition, certain vulnerabilities, like insufficient randomness in key generation, may take more effort to exploit, so breakers may skip such vulnerabilities. Finally, break-it teams may find defects that the BIBIFI infrastructure cannot automatically validate, meaning those defects will go unreported. However, with a large enough pool of break-it teams, and sufficiently general defect validations automation, we still anticipate good coverage both in breadth and depth.

Second, builders may fail to fix bugs in a manner that is in their best interests. For example, in not wanting to have a fix rejected as addressing more than one conceptual defect, teams may use several specific fixes when a more general fix would have been allowed. Additionally, teams that are out of contention for prizes may simply not participate in the fix-it phase.[3] We observed these behaviors in our contests. Both actions decrease a team's resilience score (and correspondingly increase breakers' scores). For our most recent contest, we attempted to create an incentive to fix bugs by offering prizes to participants that scale with their final score, rather than offering prizes only to winners. Unfortunately, this change in prize structure did not increase fix-it participation. We discuss fix-it behavior in more depth in Section 5.5.

Finally, there are several design points in a problem's definition and testing code that may skew results. For example, too few correctness tests may leave too many correctness bugs to be found during break-it, distracting break-it teams' attention from security issues. Too many correctness tests may leave too few bugs, meaning teams are differentiated insufficiently by general bug-finding ability. Scoring prioritizes security problems 4 to 1 over correctness problems, but it is hard to say what ratio makes the most sense when trying to maximize real-world outcomes; both higher and lower ratios could be argued. How security bugs are classified will also affect behavior; two of our contests had strong limits on the number of possible security bugs per project, while

[3]Hiding scores during the contest might help mitigate this, but would harm incentives during break-it to go after submissions with no bugs reported against them.

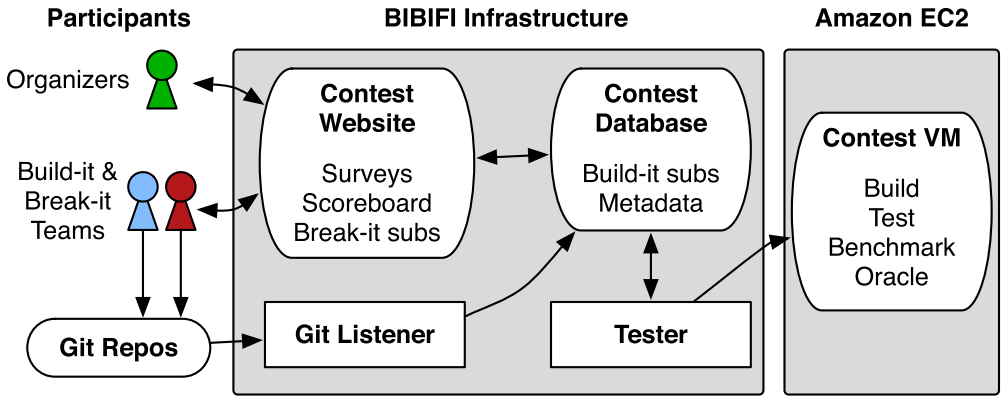


Fig. 1. Overview of BIBIFI's implementation.

the third's definition was far more (in fact, too) liberal, as discussed in Section 5.6. Finally, performance tests may fail to expose important design tradeoffs (e.g., space vs. time), affecting the ways that teams approach maximizing their ship scores. For the contests we report in this article, we are fairly comfortable with these design points. In particular, our pilot contest [57] prioritized security bugs 2 to 1 and had fewer interesting performance tests, and outcomes were better when we increased the ratio.

2.4 Implementation

Figure 1 provides an overview of the BIBIFI implementation. It consists of a web frontend, providing the interface to both participants and organizers, and a backend for testing builds and breaks. Key goals of the infrastructure are security—we do not want participants to succeed by hacking BIBIFI itself—and scalability.

Web frontend. Contestants sign up for the contest through our web application frontend, and fill out a survey when doing so, to gather demographic data potentially relevant to the contest outcome (e.g., programming experience and security training). During the contest, the web application tests build-it submissions and break-it bug reports, keeps the current scores updated, and provides a workbench for the judges for considering whether or not a submitted fix covers one bug or not.

To secure the web application against unscrupulous participants, we implemented it in ~11,500 lines of Haskell using the Yesod [10] web framework backed by a PostgreSQL [55] database. Haskell's strong type system defends against use-after-free, buffer overrun, and other memory safety-based attacks. The use of Yesod adds further automatic protection against various attacks like CSRF, XSS, and SQL injection. As one further layer of defense, the web application incorporates the information flow control framework LWeb, which is derived from LIO [61], to protect against inadvertent information leaks and privilege escalations. LWeb dynamically guarantees that users can only access their own information, as established by a mechanized proof of correctness (in Liquid Haskell [51]).

Testing backend. The backend infrastructure is used for testing during the build-it phase for correctness and performance, and during the break-it phase to assess potential vulnerabilities. It consists of ~5,500 lines of Haskell code (and a little Python).

To automate testing, we require contestants to specify a URL to a Git [5] repository (hosted on either Gitlab, Github, or Bitbucket) and shared with a designated `bibifi` username, read-only. The

backend "listens" to each contestant repository for pushes, upon which it downloads and archives each commit. Testing is then handled by a scheduler that spins up a (Docker or Amazon EC2) virtual machine that builds and tests each submission. We require that teams' code builds and runs, without any network access, in an Ubuntu Linux VM that we share in advance. Teams can request that we install additional open-source packages not present on the VM. The use of VMs supports both scalability (Amazon EC2 instances are dynamically provisioned) and security—using fresh VM instances prevents a team from affecting the results of future tests, or of tests on other teams' submissions.

All qualifying build-it submissions may be downloaded by break-it teams at the start of the break-it phase. As break-it teams identify bugs, they prepare a (JSON-based) file specifying the buggy submission along with a sequence of commands with expected outputs that demonstrate the bug. Break-it teams commit and push this file (to their Git repository). The backend uses the file to set up a test of the implicated submission to see if it indeed is a bug.

The code that tests build and break submissions differs for each contest problem. To increase modularity, we have created a problem API so that testing code run on the VM can easily be swapped out for different contest problems. Contest organizers can create their own problems by conforming to this API. The infrastructure will set up the VM and provide submission information to the problem's testing software via JSON. The problem's software runs the submission and outputs the result as JSON, which the infrastructure records and uses to update scores accordingly. Details are available in the documentation of the contest repository.

3 CONTEST PROBLEMS

This section presents the three programming problems we have developed for BIBIFI contests. These three problems were used during open competitions in 2015 and 2016, and in our own and others' undergraduate security courses since then. We discuss each problem and its specific notions of security defect, as well as how breaks exploiting such defects are automatically judged.

3.1 Secure Log

The Secure Log problem was motivated as support for an art gallery security system. Contestants write two programs. The first, `logappend`, appends events to the log; these events indicate when employees and visitors enter and exit gallery rooms. The second, `logread`, queries the log about past events. To qualify, submissions must implement two basic queries (involving the current state of the gallery and the movements of particular individuals), but they could implement two more for extra points (involving time spent in the museum, and intersections among different individuals' histories). An empty log is created by `logappend` with a given authentication token, and later calls to `logappend` and `logread` on the same log must use that token or the requests will be denied.

Here is a basic example of invocations to `logappend`. The first command creates a log file called `logfile`, because one does not yet exist, and protects it with the authentication token `secret`. In addition, it records that `Fred` entered the art gallery. Subsequent executions of `logappend` record the events of `Jill` entering the gallery and both guests entering room `1`.

```
1  $ ./logappend -K secret -A -G Fred logfile
2  $ ./logappend -K secret -A -G Jill logfile
3  $ ./logappend -K secret -A -G Fred -R 1 logfile
4  $ ./logappend -K secret -A -G Jill -R 1 logfile
```

Here is an example of `logread`, using the logfile just created. It queries who is in the gallery and what rooms they are currently in.

```
1  $ ./logread -K secret -S logfile
2  Fred
3  Jill
4  1: Fred, Jill
```

The problem states that an attacker is allowed direct access to the logfile, and yet integrity and privacy must be maintained. A canonical way of implementing the log is therefore to treat the authentication token as a symmetric key for authenticated encryption, e.g., using a combination of AES and HMAC. There are several tempting shortcuts that we anticipated build-it teams would take (and that break-it teams would exploit). For instance, one may be tempted to encrypt and sign individual log records as opposed to the entire log, thereby making `logappend` faster. But this could permit integrity breaks that duplicate or reorder log records. Teams may also be tempted to implement their own encryption rather than use existing libraries, or to simply sidestep encryption altogether. Section 4 reports several cases we observed.

A submission's performance is measured in terms of time to perform a particular sequence of operations, and space consumed by the resulting log. Correctness (and *crash*) bug reports are defined as sequences of `logread` and/or `logappend` operations with expected outputs (vetted by the oracle). Security is defined by *privacy* and *integrity*: any attempt to learn something about the log's contents, or to change them, without the using `logread` and `logappend` and the proper token should be disallowed. How violations of these properties are specified and tested is described next.

Privacy breaks. When providing a build-it submission to the break-it teams, we also included a set of log files that were generated using a sequence of invocations of that submission's `logappend` program. We generated different logs for different build-it submissions, using a distinct command sequence and authentication token for each. All logs were distributed to break-it teams without the authentication token; some were distributed without revealing the sequence of commands (the "transcript") that generated them. For these, a break-it team could submit a test case involving a call to `logread` (with the authentication token omitted) that queries the file. The BIBIFI infrastructure would run the query on the specified file with the authentication token, and if the output matched that specified by the breaker, then a privacy violation is confirmed. For example, before the break-it round, the infrastructure would run a bunch of randomly generated commands against a given team's implementation.

```
1  $ ./logappend -K secret -A -G Fred logfile
2  $ ./logappend -K secret -A -G Fred -R 816706605 logfile
```

Breakers are only given the `logfile` and not the secret token or the transcript of the commands (for privacy breaks). A breaker would demonstrate a privacy violation by submitting the following string, which matches the invocation of `./logread -K secret -S logfile`.

```
1  Fred
2  816706605: Fred
```

The system knows the breaker has successfully broken privacy, since the breaker is able to present confidential information without knowing the secret token. In practice, the transcript of commands is significantly longer and a random secret is used.

Integrity breaks. For about half of the generated log files, we also provided the transcript of the `logappend` operations used to generate the file. A team could submit a test case specifying the name of the log file, the contents of a corrupted version of that file, and a `logread` query over it (without the authentication token). For both the specified log file and the corrupted one, the BIBIFI infrastructure would run the query using the correct authentication token. An integrity violation

is detected if the query command produces a non-error answer for the corrupted log that differs from the correct answer (which can be confirmed against the transcript using the oracle).

This approach to determining privacy and integrity breaks has the benefit and drawback that it does not reveal the *source* of the issue, only that there is (at least) one, and that it is exploitable. As such, we only count up to one integrity break and one privacy break against the score of each build-it submission, even if there are multiple defects that could be exploited to produce privacy/integrity violations (since we could not automatically tell them apart).

3.2 ATM

The ATM problem asks builders to construct two communicating programs: `atm` acts as an ATM client, allowing customers to set up an account, and deposit and withdraw money; `bank` is a server that tracks client bank balances and processes their requests, received via TCP/IP. `atm` and `bank` should only permit a customer with a correct *card file* to learn or modify the balance of their account, and only in an appropriate way (e.g., they may not withdraw more money than they have). In addition, `atm` and `bank` should only communicate if they can authenticate each other. They can use an *auth file* for this purpose; it will be shared between the two via a trusted channel unavailable to the attacker.[4] Since the `atm` is communicating with `bank` over the network, a "man in the middle" (MITM) could observe and modify exchanged messages, or insert new messages. The MITM could try to compromise security despite not having access to auth or card files. Such compromise scenarios are realistic, even in 2018 [63].

Here is an example run of the `bank` and `atm` programs.

```
1  $ ./bank -s bank.auth &
```

This invocation starts the `bank` server, which creates the file `bank.auth`. This file will be used by the `atm` client to authenticate the `bank`. The `atm` is started as follows:

```
1  $ ./atm -s bank.auth -c bob.card -a bob -n 1000.00
2  {"account":"bob","initial_balance":1000}
```

The client initiates creation of a new account for user `bob` with an initial balance of $1,000. It also creates a file `bob.card` that is used to authenticate `bob` (this is basically Bob's PIN) from here on. A receipt of the transaction from the server is printed as JSON. The `atm` client can now use `bob`'s card to perform further actions on his account. For example, this command withdraws $63.10 from `bob`'s account:

```
1  $ ./atm -s bank.auth -c bob.card -a bob -w 63.10
2  {"account":"bob","withdraw":63.1}
```

A canonical way of implementing the `atm` and `bank` programs would be to use public key-based authenticated and encrypted communications. The auth file is used as the `bank`'s public key to ensure that key negotiation initiated by the `atm` is with the `bank` and not a MITM. When creating an account, the card file should be a suitably large random number, so that the MITM is unable to feasibly predict it. It is also necessary to protect against replay attacks by using nonces or similar mechanisms. As with Secure Log, a wise approach would be use a library like OpenSSL to implement these features. Both good and bad implementations are discussed further in Section 4.

Build-it submissions' performance is measured as the time to complete a series of benchmarks involving various `atm`/`bank` interactions.[5] Correctness (and *crash*) bug reports are defined as

[4] In a real deployment, this might be done by "burning" the auth file into the ATM's ROM prior to installing it.
[5] The transcript of interactions is always serial, so there was no motivation to support parallelism for higher throughput.

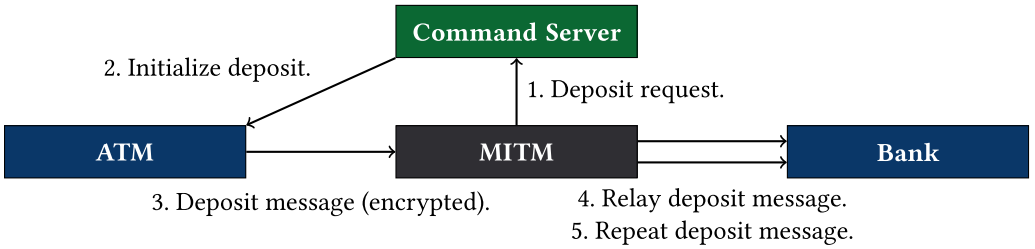


Fig. 2. MITM replay attack.

sequences of `atm` commands where the targeted submission produces different outputs than the oracle (or crashes). Security defects are specified as follows.

Integrity breaks. Integrity violations are demonstrated using a custom MITM program that acts as a proxy: It listens on a specified IP address and TCP port, and accepts a connection from the `atm` while connecting to the `bank`. We provided a Python-based proxy as a starter MITM; it forwards communications between the endpoints after establishing the connection. A breaker's MITM would modify this baseline behavior, observing and/or modifying messages sent between `atm` and `bank`, and perhaps dropping messages or initiating its own.

To demonstrate an integrity violation, the MITM will send requests to a *command server*. It can tell the server to run inputs on the `atm` and it can ask for the card file for any account whose creation it initiated. Eventually the MITM will declare the test complete. At this point, the same set of `atm` commands is run using the oracle's `atm` and `bank` *without the MITM*. This means that any messages that the MITM sends directly to the target submission's `atm` or `bank` will not be replayed/sent to the oracle. If the oracle and target both complete the command list without error, but they differ on the outputs of one or more commands, or on the balances of accounts at the bank whose card files were not revealed to the MITM during the test, then there is evidence of an integrity violation.

As an example (based on a real attack we observed), consider a submission that uses deterministic encryption without nonces in messages. The MITM could direct the command server to deposit money from an account, and then replay the message it observes. When run on the vulnerable submission, this would credit the account twice. But when run on the oracle without the MITM, no message is replayed, leading to differing final account balances. A correct submission would reject the replayed message, which would invalidate the break. This example is illustrated in Figure 2.

Privacy breaks. Privacy violations are also demonstrated using a MITM. In this case, at the start of a test the command server will generate two random values, *amount* and *account name*. If by the end of the test no errors have occurred and the attacker can prove it knows the actual value of either secret (by sending a command that specifies it), then the break is considered successful. Before demonstrating knowledge of the secret, the MITM can send commands to the and server with a *symbolic* reference to *amount* and *account name*; the server will fill in the actual secrets before forwarding these messages. The command server does not automatically create a secret account or an account with a secret balance; it is up to the breaker to do that (referencing the secrets symbolically when doing so).

As an example, suppose the target does not encrypt exchanged messages. Then a privacy attack might be for the MITM to direct the command server to create an account whose balance contains the secret amount. Then the MITM can observe an unencrypted message sent from `atm` to `bank`; this message will contain the actual amount filled in by the command server. The MITM can then send its guess to the command server showing that it knows the amount.

```
<prog>     ::= as principal p password s do \n <cmd> ***
<cmd>      ::= exit \n | return <expr> \n | <prim_cmd> \n <cmd>
<expr>      ::=  <value> | [] |  <fieldvals>
<fieldvals> ::=  x = <value> | x = <value> , <fieldvals>
<value>    ::=  x | x . y | s
<prim_cmd> ::= create principal p s
             | change password p s
             | set x = <expr>
             | append to x with <expr>
             | local x = <expr>
             | foreach y in x replacewith <expr>
             | set delegation <tgt> q <right> -> p
             | delete delegation <tgt> q <right> -> p
             | default delegator = p
<tgt>      ::= all | x
<right>    ::= read | write | append | delegate
```

Fig. 3. Grammar for the Multiuser DB command language as BNF. Here, x and y represent arbitrary variables; p and q represent arbitrary principals; and s represents an arbitrary string. Commands submitted to the server should match the non-terminal <prog>.

As with the Secure Log problem, we cannot tell whether an integrity or privacy test is exploiting the same underlying weakness in a submission, so we only accept one violation of each category against each submission.

Timeouts and denial of service. One difficulty with our use of a breaker-provided MITM is that we cannot reliably detect bugs in `atm` or `bank` implementations that would result in infinite loops, missed messages, or corrupted messages. This is because such bugs can be simulated by the MITM by dropping or corrupting messages it receives. Since the builders are free to implement any protocol they like, our auto-testing infrastructure cannot tell if a protocol error or timeout is due to a bug in the target or due to misbehavior of the MITM. As such, we conservatively disallow any MITM test run that results in the target `atm` or `bank` hanging (timing out) or returning with a protocol error (e.g., due to a corrupted packet). This means that flaws in builder implementations might exist but evidence of those bugs might not be realizable in our testing system.

3.3 Multiuser DB

The Multiuser DB problem requires builders to implement a server that maintains a multi-user key-value store, where users' data is protected by customizable access control policies. The data server accepts queries written in a text-based command language delivered over TCP/IP (we assume the communication channel is trusted, for simplicity). Each program begins by indicating the querying user, authenticated with a password. It then runs a sequence of commands to read and write data stored by the server, where data values can be strings, lists, or records. The full grammar of the command language is shown in Figure 3 where the start symbol (corresponding to a client command) is <prog>. Accesses to data may be denied if the authenticated user lacks the necessary permission. A user can delegate permissions, like reading and writing variables, to other principals. If running the command program results in a security violations or error, then all of its effects will be rolled back.

Here is an example run of the data server. To start, the server is launched and listens on TCP port 1024.

```
1  $ ./server 1024 &
```

Next, the client submits the following program.

```
as principal admin password "admin" do
  create principal alice "alices_password"
  set msg = "Hi Alice. Good luck!"
  set delegation msg admin read -> alice
  return "success"
***
```

The program starts by declaring it is running on behalf of principlal `admin`, whose password is `"admin"`. If authentication is successful, then the program creates a new principal `alice`, creates a new variable `msg` containing a string, and delegates read permission on the variable to Alice. The server sends back a transcript of the successful commands to the client, in JSON format:

```
{"status":"CREATE_PRINCIPAL"}
{"status":"SET"}
{"status":"SET_DELEGATION"}
{"status":"RETURNING","output":"success"}
```

Next, suppose Alice sends the following program, which simply logs in and reads the `msg` variable:

```
as principal alice password "alices_password" do
  return msg
***
```

The server response indicates the result:

```
{"status":"RETURNING","output":"Hi Alice. Good luck!"}
```

The data server is implemented by writing a parser to read the input command programs. The server needs a store to keep the value of each variable as well as an access control list that tracks the permissions for the variables. It also needs to keep track of delegated permissions, which can form chains; e.g., Alice could delegate read permission to all of her variables to Bob, who could then delegate permission to read one of those variables to Charlie. If when executing the program a security violation or other error occurs (e.g., reading a variable that does not exist), then the server needs to roll back its state to what it was prior to processing the input program. All responses back to the client are encoded as JSON.

Scoring. A data server's performance is measured in elapsed runtime to process sequences of programs. Correctness (and *crash*) violations are demonstrated by providing a sequence of command programs where the data server's output differs from that of the oracle (or the implementation crashes). Security violations can be to data privacy, integrity, or availability, by comparing the behavior of the target against that of an oracle implementation.

Privacy breaks. A privacy violation occurs when the oracle would deny a request to read a variable, but the target implementation allows it. Consider the following example where a variable, `secret`, is created, but Bob is not allowed to read it.

```
as principal admin password "admin" do
  create principal bob "bobs_password"
  set secret = "Super secret"
  return "success"
***

```

```
{"status":"CREATE_PRINCIPAL"}
{"status":"SET"}
{"status":"RETURNING","output":"success"}
```

Now Bob attempts to read the `secret` variable with the following query.

```
as principal bob password "bobs_password" do
  return secret
***
```

Bob does not have permission to read `secret`, so the oracle returns `{"status": "DENIED"}`. If the implementation returns the `secret` contents of `{"status": "RETURNING","output": "Super secret"}`, then we know a confidentiality violation has occurred.

Integrity breaks. Integrity violations are demonstrated in a similar manner, but occur when unprivileged users modify variables they do not have permission to write to. With the example above, the variable `secret`, is created, but Bob is not allowed to write to it. Now Bob attempts to write to `secret` with the following query.

```
as principal bob password "bobs_password" do
  set secret = "Bob's grade is an A!"
  return "success"
***
```

Bob does not have write permission on `secret`, so the oracle returns `{"status":"DENIED"}`. If the implementation returns the following, then an integrity violation has been demonstrated.

```
{"status":"SET"}
{"status":"RETURNING","output":"success"}
```

Availability breaks. Unlike the ATM problem, we are able to assess availability violations for Multiuser DB (since we are not using a MITM). In this case, a security violation is possible when the server implementation is unable to process legal command programs. This is demonstrated when the target incorrectly denies a program by reporting an error, but the oracle successfully executes the program. Availability security violations also happen when the server implementation fails to respond to an input program within a fixed period of time.

Unlike the other two problems, the Multiuser DB problem does not place a limit on the number of security breaks submitted. In addition, the overall bug submission limit is reduced to 5, as opposed to 10 for the other two problems. Recall that for Secure Log and ATM, a break constitutes direct evidence that a vulnerability has been exploited, but not *which* vulnerability, if more than one is present. As such, if a build-it team were to fix a vulnerability during the fix-it phase, doing so would not shed light on which breaks derived from that vulnerability, vs. others. The contest thus limits break-it teams to one break each for confidentiality and integrity, per target team. (See Section 3.1 and Section 3.2.) For Multiuser DB, a security vulnerability is associated with a test run, so a fix of that vulnerability *will* unify all breaks that exploit that vulnerability, just as correctness fixes do. That means we need not impose a limit on them. The consequences of this design are discussed in Section 5.

4 BUILD-IT SUBMISSIONS: SUCCESSES AND FAILURES

After running a BIBIFI contest, we have all of the code written by the build-it teams, and the bug reports submitted by the break-it teams. Looking at these artifacts, we can get a sense of what build-it teams did right, and what they did wrong. This section presents a sample of failure and success stories, while Section 5 presents a broader statistical analysis that suggests overall trends.

A companion paper [69] provides a more detailed review of the types of vulnerabilities based on an extensive, in-depth qualitative analysis of submitted code.

4.1 Failure Stories

The failure modes for build-it submissions are distributed along a spectrum ranging from "failed to provide any security at all" to "vulnerable to extremely subtle timing attacks." This is interesting, because a similar dynamic is observed in the software marketplace today.

Secure Log. Many implementations of the Secure Log problem failed to use encryption or authentication codes, presumably because the builders did not appreciate the need for them. Exploiting these design flaws was trivial for break-it teams. Sometimes log data was written as plain text, other times log data was serialized using the Java object serialization protocol.

One break-it team discovered a privacy flaw that they could exploit with at most fifty probes. The target submission truncated the authentication token (i.e., the key) so that it was vulnerable to a brute force attack.

Some failures were common across Secure Log implementations: if an implementation used encryption, it might not use authentication. If it used authentication, then it would authenticate records stored in the file individually, not globally. The implementations would also relate the ordering of entries in the file to the ordering of events in time, allowing for an integrity attack that changes history by re-ordering entries in the file.

There were five crashes due to memory errors, and they all occurred in C/C++ submissions. We examined two of the crashes and confirmed that they were exploitable. The first was a null pointer dereference, and the other was a buffer overflow from the use of `strcpy`.

ATM. The ATM problem allows for interactive attacks (not possible for the log), and the attacks became cleverer as implementations used cryptographic constructions incorrectly. One implementation used cryptography, but implemented RC4 from scratch and did not add any randomness to the key or the cipher stream. An attacker observed that the ciphertext of messages was distinguishable and largely unchanged from transaction to transaction, and was able to flip bits in a message to change the withdrawn amount.

Another implementation used encryption with authentication, but did not use randomness; as such error messages were always distinguishable from success messages. An attack was constructed against this implementation where the attack leaked the bank balance by observing different withdrawal attempts, distinguishing the successful from failed transactions, and performing a binary search to identify the bank balance given a series of withdraw attempts.

Some failures were common across ATM problem implementations. Many implementations kept the key fixed across the lifetime of the `bank` and `atm` programs and did not use a nonce in the messages. This allowed attackers to replay messages freely between the `bank` and the `atm`, violating integrity via unauthorized withdrawals. Several implementations used encryption but without authentication. (This sort of mistake has been observed in real-world ATMs, as has, amazingly, a complete lack of encryption use [63].) These implementations used a library such as OpenSSL, the Java cryptographic framework, or the Python pycrypto library to have access to a symmetric cipher such as AES, but either did not use these libraries at a level where authentication was provided in addition to encryption, or they did not enable authentication.

Multiuser DB. The Multiuser DB problem asks participants to consider a complex logical security problem. In this scenario, the specification was much more complicated. All but one team developed a system of home-grown access control checks. This led to a variety of failures when participants were unable to cover all possible security edge cases.

In some instances, vulnerabilities were introduced, because they did not properly implement the specification. Some participants hardcoded passwords making them easily guessable by an attacker. Other participants did not include checks for the delegation command to ensure that the principal had the right to delegate along with the right they were trying to delegate.

Other participants failed to consider the security implications of their design decisions when the specification did not provide explicit instructions. For example, many of the participants did not check the delegation chain back to its root. Therefore, once a principal received an access right, they maintained this right even if it no longer belonged to the principal that delegated it to them.

There were two crashes targeting Multiuser DB implementations. Both were against C/C++ submissions. We inspected one crash and determined it was caused by code in the parser that dereferenced and executed an invalid (non-null) pointer.

Finally, other teams simply made errors when implementing the access control logic. In some cases, these mistakes introduced faulty logic into the access control checks. One team made a mistake in their control flow logic such that if a principal had no delegated rights, the access control checks were skipped—because a lookup error would have occurred. In other cases, these mistakes led to uncaught runtime errors that allowed the attacker to kill the server, making it unavailable to other users.

4.2 Success Stories

In contrast to the broken submissions, successful submissions followed understood best practices. For example, submissions made heavy use of existing high-level cryptographic libraries with few "knobs" that allow for incorrect usage [18]. Similarly, successful submissions limited the size and location of security-critical code [50].

ATM and Secure Log. One implementation of the ATM problem, written in Python, made use of the SSL PKI infrastructure. The implementation used generated SSL private keys to establish a root of trust that authenticated the `atm` program to the `bank` program. Both the `atm` and `bank` required that the connection be signed with the certificate generated at runtime. Both the `bank` and the `atm` implemented their communication protocol as plain text then wrapped in HTTPS. To find bugs in this system, other contestants would need to break the security of OpenSSL.

Another implementation, written in Java, used the NaCl library. This library intentionally provides a very high level API to "box" and "unbox" secret values, freeing the user from dangerous choices. As above, to break this system, other contestants would need to break NaCl first.

A Java-based implementation of the Secure Log problem used the BouncyCastle library's high-level API to construct a valid encrypt-then-MAC scheme over the entire log file. BouncyCastle allowed them to easily authenticate the whole log file, protecting them from integrity attacks that swapped the order of encrypted binary data in the log.

Multiuser DB. The most successful solutions for the Multiuser DB problem were localized access control logic checks to a single function with a general interface, rather repeating checking code for each command that needed it. Doing so reduced the likelihood of a mistake. One of the most successful teams used a fairly complex graphical representation of access control rules, but by limiting the number of places this graph was manipulated they could efficiently and correctly check access rights without introducing vulnerabilities.

5 QUANTITATIVE ANALYSIS

This section quantitatively analyzes data we gathered from our 2015 and 2016 contests.[6] We consider participants' performance in each contest phase, identifying factors that contribute to high

[6]We also ran a contest during Fall'14 [57] but exclude its data due to differences in how it was administered.

scores after the build-it round, resistance to breaking by other teams, and strong performance as breakers.

We find that on average, teams that program using statically typed languages are 11× less likely to have security bugs identified in their code compared to those using C and C++. Success in breaking, and particularly in identifying security bugs in other teams' code, is correlated with having more team members, as well as with participating successfully in the build-it phase (and therefore having given thought to how to secure an implementation). The use of advanced techniques like fuzzing and static analysis was dropped from the final model, indicating that their effect was not statistically significant. We note that such tools tend to focus on bugs, like memory errors and taint/code injection attacks, that were rare in our contests (per Section 4). Overall, integrity bugs were far more common than privacy bugs or crashes. The contests that used the ATM problem and the Multiuser DB problem were associated with more security bugs than the Secure Log contest.

5.1 Data Collection

For each team, we collected a variety of observed and self-reported data. When signing up for the contest, teams reported standard demographics and features such as coding experience and programming language familiarity. After the contest, each team member optionally completed a survey about their performance. In addition, we extracted information about lines of code written, number of commits, and so on, from teams' Git repositories.

Participant data was anonymized and stored in a manner approved by our institution's human-subjects review board. Participants consented to have data related to their activities collected, anonymized, stored, and analyzed. A few participants did not consent to research involvement, so their personal data was not used in the data analysis.

5.2 Analysis Approach

To examine factors that correlated with success in building and breaking, we apply regression analysis. Each regression model attempts to explain some outcome variable using one or more measured factors. For most outcomes, such as participants' scores, we can use ordinary linear regression, which estimates how many points a given factor contributes to (or takes away from) a team's score. To analyze binary outcomes, such as whether or not a security bug was found, we apply logistic regression, which estimates how each factor impacts the likelihood of an outcome.

We consider many variables that could potentially impact teams' results. To avoid over-fitting, we select as potential factors those variables that we believe are of most interest, within acceptable limits for power and effect size. As we will detail later, we use the same factors as the analysis in our earlier conference paper [56], plus one more, which identifies participation in the added contest (Multiuser DB). The impact of the added data on the analysis, compared to the analysis in the earlier paper, is considered in Section 5.8. We test models with all possible combinations of our chosen potential factors and select the model with the minimum Akaike Information Criterion (AIC) [21]. The final models are presented.

Each model is presented as a table with each factor as well as the p-value for that factor. Significant p-values (<0.05) are marked with an asterisk. Linear models include the coefficient relative to the baseline factor and the 95% confidence interval. Logistic models also include the exponential coefficient and the 95% confidence interval for the exponential coefficient. The exponential coefficient indicates how many times more likely the measured result occurs relative to the baseline factor.

We describe the results of each model below. This was not a completely controlled experiment (e.g., we do not use random assignment), so our models demonstrate correlation rather than causation. Our observed effects may involve confounds, and many factors used as independent variables

Table 2. Contestants, by Self-reported Country

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

in our data are correlated with each other. This analysis also assumes that the factors we examine have linear effect on participants' scores (or on likelihood of binary outcomes); while this may not be the case in reality, it is a common simplification for considering the effects of many factors. We also note that some of the data we analyze is self-reported, so may not be entirely precise (e.g., some participants exaggerating about which programming languages they know); however, minor deviations, distributed across the population, act like noise and have little impact on the regression outcomes.

5.3 Contestants

We consider three contests offered at different times:

Secure Log: We held one contest using the Secure Log problem (Section 3.1) during May–June 2015 as the capstone to a Cybersecurity MOOC sequence.[7] Before completing in the capstone, participants passed courses on software security, cryptography, usable security, and hardware security.

ATM: During Oct.–Nov. 2015, we offered the ATM problem (Section 3.2) as two contests simultaneously, one as a MOOC capstone, and the other open to U.S.-based graduate and undergraduate students. We merged the contests after the build-it phase, due to low participation in the open contest. MOOC and open participants were ranked independently to determine grades and prizes.

Multiuser DB: In Sep.–Oct. 2016, we ran one constest offering the Multiuser DB problem (Section 3.3) open to both MOOC capstone participants as well as graduate and undergraduate students.

The U.S. was the most represented country in our contestant pool, but was not the majority. There was also representation from developed countries with a reputation both for high technology and hacking acumen. Details of the most popular countries of origin can be found in Table 2, and additional information about contestant demographics is presented in Table 3. In total, 156 teams participated in either the build-it or break-it phases, most of which participated in both.

5.4 Ship Scores

We first consider factors correlating with a team's *ship* score, which assesses their submission's quality before it is attacked by the other teams (Section 2.1). This data set contains all 130 teams from the Secure Log, ATM, and Multiuser DB contests that qualified after the build-it phase. The contests have nearly the same number of correctness and performance tests, but different numbers of participants. We set the constant multiplier M to be 50 for the contests, which effectively normalizes the scores (see Section 2.2).

Model setup. To ensure enough statistical power to find meaningful relationships, our modeling was designed for a prospective effect size roughly equivalent to Cohen's *medium* effect heuristic, $f^2 = 0.15$ [25]. An effect of this size corresponds to a coefficient of determination $R^2 = 0.13$, suggesting we could find an effect if our model can explain at least 13% of the variance in the

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

Table 3. Demographics of Contestants from Qualifying Teams

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%

Some participants declined to specify gender.

Table 4. Factors and Baselines for Build-it Models

Factor	Description	Baseline
Contest	Secure Log, ATM, or Multiuser DB contest.	Secure Log
# Team members	A team's size.	—
Knowledge of C	The fraction of team members who know C or C++.	—
# Languages known	Number of programming languages team members know.	—
Coding experience	Average years of programming experience.	—
Language category	C/C++, statically typed, or dynamically typed language.	C/C++
Lines of code	Lines of code count for the team's final submission.	—
MOOC	If the team was participating in the MOOC capstone.	non-MOOC

outcome variable. We report the observed coefficient of determination for the final model with the regression results below.

As mentioned above, we reuse the factors chosen for the analysis in our earlier paper [56]. Their number was guided by a power analysis of the contest data we had at the time, which involved the $N = 101$ build-it teams that participated in Secure Log and ATM. With an assumed power of 0.75, the power analysis suggested we limit the covariate factors used in our model to nine degrees of freedom, which yields a prospective $f^2 = 0.154$. With the addition of Multiuser DB data, we add one more factor, which is choice of Multiuser DB as an option for which contest the submission belongs to. This adds a 10th degree of freedom, as well as 29 additional teams for a total $N = 130$. At 0.75 power, this yields a prospective $f^2 = 0.122$, which is better than in the earlier paper's analysis.

We selected the factors listed in Table 4. *Knowledge of C* is included as a proxy for comfort with low-level implementation details, a skill often viewed as a prerequisite for successful secure building or breaking. *# Languages known* is how many unique programming languages team members collectively claim to know (see the second to last row of Table 3). For example, on a two-member team where member A claims to know C++, Java, and Perl and member B claims to know Java, Perl, Python, and Ruby, the language count would be 5. *Language category* is the "primary" language category we manually identified in each team's submission. Languages were categorized as either C/C++, statically typed (e.g., Java, Go, but not C/C++), or dynamically typed (e.g., Perl, Python).

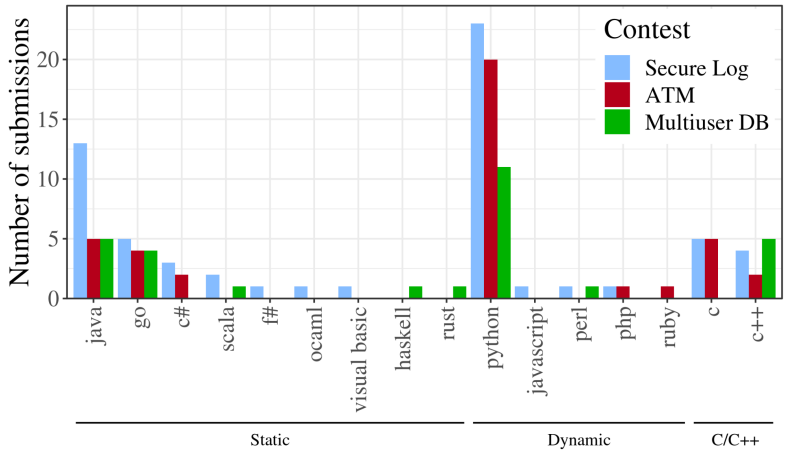


Fig. 4. The number of build-it submissions in each contest, organized by primary programming language used. The languages are grouped by category.

Table 5. Final Linear Regression Model of Teams' Ship Scores, Indicating How Many Points Each Selected Factor Adds to the Total Score

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

$R^2 = 0.232$.

Precise category allocations, and total submissions for each language, segregated by contest, are given in Figure 4.

Results. The final model (Table 5) with $R^2 = 0.232$ captures almost $\frac{1}{4}$ of the variance. We find this number encouraging given how relatively uncontrolled the environment is and how many contributing, but unmeasured, factors there could be. Our regression results indicate that ship score is strongly correlated with language choice. Teams that programmed in C or C++ performed on average 133 and 112 points better than those who programmed in dynamically typed or statically typed languages, respectively. Figure 5 illustrates that while teams in many language categories performed well in this phase, only teams that did not use C or C++ scored poorly.

The high scores for C/C++ teams could be due to better scores on performance tests and/or due to implementing optional features. We confirmed the main cause is the former. Every C/C++ team for the Secure Log contest implemented all optional features, while six teams in the other categories implemented only six of ten and one team implemented none; the ATM contest offered no optional features; for the Multiuser DB contest, four C/C++ teams implemented all optional features while one C/C++ team implemented five of nine. We artificially increased the scores of all teams as if they had implemented all optional features and reran the regression model. In the

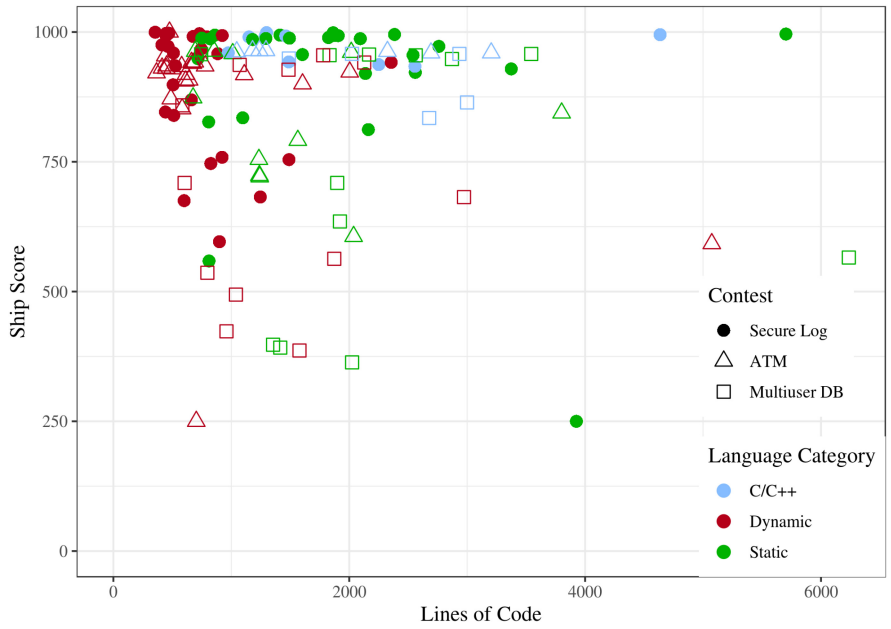


Fig. 5. Each team's ship score, compared to the lines of code in its implementation and organized by language category and contest. Using C/C++ correlates with a higher ship score.

resulting model, the difference in coefficients between C/C++ and the other language categories dropped only slightly. This indicates that the majority of improvement in C/C++ ship score comes from performance.

The number of languages known by a team is not quite statistically significant, but the confidence interval in the model suggests that each programming language known increases ship scores by between 0 and 12 points. Intuitively, this makes sense, since contestants that know more languages have more programming experience and have been exposed to different paradigms.

Lines of code is also not statistically significant, but the model hints that each additional line of code in a team's submission is associated with a minor drop in ship score. Based on our qualitative observations (see Section 4), we hypothesize this may relate to more reuse of code from libraries, which frequently are not counted in a team's LOC (most libraries were installed directly on the VM, not in the submission itself). We also found that, as further noted above, submissions that used libraries with more sophisticated, lower-level interfaces tended to have more code and more mistakes; i.e., more steps took place in the application (more code) but some steps were missed or carried out incorrectly (less secure/correct). Figure 5 shows that LOC is (as expected) associated with the category of language being used. While LOC varied widely within each language type, dynamic submissions were generally shortest, followed by static submissions and then those written in C/C++ (which has the largest minimum size).[8]

5.5 Resilience

Now, we turn to measures of a build-it submission's quality, starting with *resilience*. Resilience is a non-positive score that derives from break-it teams' bug reports, which are accompanied by test

[8]Our earlier model for the Secure Log and ATM contests found that lines of code was actually statistically significant. We discuss this further in Section 5.8.

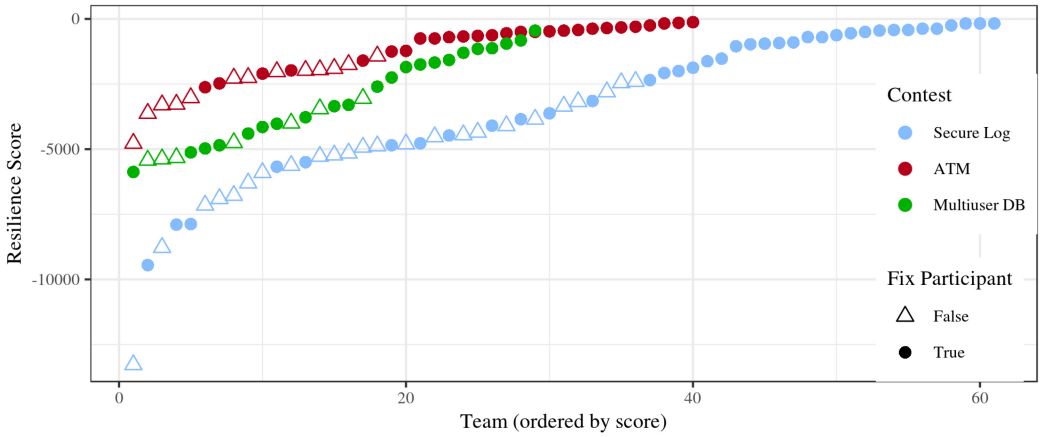


Fig. 6. Final resilience scores, ordered by team, and plotted for each contest problem. Build-it teams who did not bother to fix bugs generally had lower scores.

Table 6. Break-it Teams in Each Contest Submitted Bug Reports, Which Were Judged by the Automated Oracle

	Secure Log	ATM	Multiuser DB
Bug reports submitted	24,796	3,701	3,749
Bug reports accepted	9,482	2,482	2,046
Fixes submitted	375	166	320
Bug reports addressed by fixes	2,252	966	926

Build-it teams then submitted fixes, each of which could potentially address multiple bug reports.

cases that prove the presence of defects. The overall build-it score is the sum of ship score, just discussed, and resilience. Builders may increase the resilience component during the fix-it phase, as fixes prevent double-counting bug reports that identify the same defect (see Section 2.1).

Unfortunately, upon studying the data, we found that a large percentage of build-it teams opted not to fix any bugs reported against their code, forgoing the scoring advantage of doing so. We can see this in Figure 6, which graphs the resilience scores (Y-axis) of all teams, ordered by score, for the three contests. The circles in the plot indicate teams that fixed at least one bug, whereas the triangles indicate teams that fixed no bugs. We can see that, overwhelmingly, the teams with the lower resilience scores did not fix any bugs. Table 6 digs a little further into the situation. It shows that of the bug reports deemed acceptable by the oracle (the second row), submitted fixes (row 3) addressed only 23% of those from the Secure Log contest, 38% of those from the ATM contest, and 45% of those from the Multiuser DB contest (row 4 divided by row 2). It turns out that when counting only "active" fixers who fixed at least one bug, these averages were 56.9%, 72.5%, and 64.6%, respectively.

Incentivizing fixing. This situation is disappointing, as we cannot treat resilience score as a good measure of code quality (when added to ship score). After the first two contests, we hypothesized that participants were not sufficiently incentivized to fix bugs, for two reasons. First, teams that were sufficiently far from the lead may have chosen to fix no bugs, because winning was unlikely. Second, for MOOC students, once a minimum score is achieved they were assured to pass; it may be that fixing (many) bugs was unnecessary for attaining this minimum score.

Table 7. Final Logistic Model Measuring Log-likelihood of the Discovery of a Security Bug in a Team's Submission

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 $R^2 = 0.619$.

We attempted to more strongly incentivize all teams to fix (duplicated) bugs by modifying the prize structure for the Multiuser DB contest. Instead of only giving away prizes to top teams, non-MOOC participants could still win monetary prizes if they scored outside of third place. Placements were split into different brackets, and one team from each bracket was randomly selected to receive a prize. Prizes increased based on bracket position (e.g., the fourth and fifth place bracket winner received $500, while the sixth and seventh place bracket winner received $375). Our hope was that builders would submit fixes to bump themselves into a higher bracket, which would have a larger payout. Unfortunately, it does not appear that fix participation increased for non-MOOC participants for the Multiuser DB contest. To confirm this, we ran a linear regression model, but according to the model, incentive structure was not a factor in fix participation. The model did confirm that teams with a higher score at the end of break-it fixed a greater percentage of the bugs against them.

Additionally, we randomly sampled 60% of Multiuser DB teams to identify the types of vulnerabilities they chose to fix. We manually analyzed each break to determine the underlying vulnerability to determine whether the expected fix difficulty impacted team decisions. We did not observe any clear trend in the vulnerabilities teams chose to fix, with all vulnerability types both fixed by some teams and not fixed by others. Instead, we found teams most often made a binary decision, choosing to fix all (38%) or none (38%) of their vulnerabilities. The remaining teams only slightly strayed from a binary choice by either fixing all but one vulnerability (16%) or only one vulnerability (8%).

5.6 Presence of Security Bugs

While resilience score is not sufficiently meaningful, a useful alternative is the likelihood that a build-it submission contains a security-relevant bug; by this, we mean any submission against which at least one crash, privacy, integrity, or availability defect is demonstrated. In this model, we used logistic regression over the same set of factors as the ship model.

Table 7 lists the results of this logistic regression; the coefficients represent the change in log likelihood associated with each factor. Negative coefficients indicate lower likelihood of finding a security bug. For categorical factors, the exponential of the coefficient (*exp(coef)*) indicates how strongly that factor being true affects the likelihood relative to the baseline category.[9] For numeric

[9]In cases (such as the ATM contest) where the rate of security bug discovery is close to 100%, the change in log likelihood starts to approach infinity, somewhat distorting this coefficient upwards.

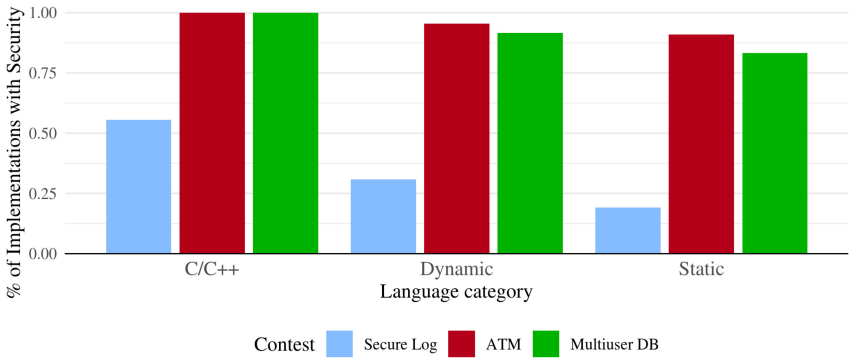


Fig. 7. The fraction of teams in whose submission a security bug was found, by contest and language category.

factors, the exponential indicates how the likelihood changes with each unit change in that factor. R^2 as traditionally understood does not make sense for a logistic regression. There are multiple approximations proposed in the literature, each of which have various pros and cons. We present Nagelkerke ($R^2 = 0.619$), which suggests the model explains an estimated 61% of variance [45].

ATM implementations were far more likely than Secure Log implementations to have a discovered security bug.[10] We hypothesize this is due to the increased security design space in the ATM problem as compared to the Secure Log problem. Although it is easier to demonstrate a security error in the Secure Log problem, the ATM problem allows for a much more powerful adversary (the MITM) that can interact with the implementation; breakers often took advantage of this capability, as discussed in Section 4.

Multiuser DB implementations were 31× as likely as Secure Log implementations to have a discovered security bug. We hypothesize this is due to increased difficulty in implementing a custom access control system. There are limited libraries available that directly provide the required functionality, so contestants needed to implement access control manually, leaving more room for error. For the Secure Log problem, builders could utilize cryptographic libraries to secure their applications. In addition, it was potentially easier for breakers to discover attacks with the Multiuser DB problem, since they could reuse break tests against multiple build submissions.[11]

The model also shows that C/C++ implementations were more likely to contain an identified security bug than either static- or dynamic-language implementations. For static languages, this effect is significant and indicates that—assuming all other features are the same—a C/C++ program was about 11× (that is, 1/0.089 given in Table 7[12]) more likely to contain an identified bug. This effect is clear in Figure 7, which plots the fraction of implementations that contain a security bug, broken down by language type and contest problem. Of the 21 C/C++ submissions (see Figure 4),

[10]This coefficient (corresponding to 103×) is somewhat exaggerated (see prior footnote), but the difference between contests is large and significant.

[11]One caveat here is that a quirk of the problem definition permitted breakers to escalate correctness bugs into security problems by causing the state of Multiuser DB submissions to become out of sync with the oracle implementation, and behave in a way that seemed to violate availability. We only realized after the contest was over that these should have been classified as correctness bugs. For the data analysis, we retroactively reclassified these bugs as correctness problems. Had they been classified properly during the contest, break-it team behavior might have changed, i.e., to spend more time hunting proper security defects.

[12]Here, we use the inverse of the exponential coefficient for **Statically Typed**, because we are describing the relationship between variables in the opposite direction than as presented in the table, i.e., the baseline **C/C++** in comparison to **Statically Typed** as opposed to **Statically Typed** in comparison to baseline **C/C++**.

Table 8. The Number and Percentage of Teams that had Different Types of Security Bugs by Language Category

Language Category	Integrity	Confidentiality	Integrity, Confidentiality, or Availability (Multiuser DB)	Crash
C/C++	9/43%	4/19%	5/24%	7/33%
Dynamic	27/45%	17/28%	11/18%	0/0%
Static	15/31%	10/20%	10/20%	0/0%

Percentages are relative to total submissions in that language category, across all contests. Integrity, confidentiality, and availability bugs were not distinguished for the Multiuser DB problem during that contest. We group them in their own column.

17 of them had a security bug: 5/9 for the Secure Log contest, 7/7 for the ATM contest, and 5/5 for the Multiuser DB contest. All five of the buggy implementations from the Secure Log contest had a crash defect, and crashes were the only security-related problem for three of them; none of the ATM implementations had crash defects; two of the Multiuser DB C/C++ submissions had crash defects. All crash defects were due to violation of memory safety. More details about the crashes are presented in Section 4.1. Table 8 breaks down the number and percentage of teams that had different categories of security bugs.

The model shows four factors that played a role in the outcome, but not in a statistically significant way: using a dynamically typed language, lines of code of an implementation, developer knowledge of C, and number of team members. We see the effect of the first in Figure 7. We note that the number of team members is just outside the threshold of being significant. This suggests that an implementation is 1.4× (1/0.705) less likely to have a security bug present for each team member.

Finally, we note that MOOC participation was not included in our final model, indicating that (this kind of) security education did not have a significant effect in the outcome. Prior research [48] similarly did not find a significant effect of education in secure programming contexts. Our previous work investigating differences between experts (hackers) and non-experts (software testers) suggests improvements in vulnerability finding skill are driven by direct experiences with a variety of vulnerabilities (e.g., discovering them, or being shown specific examples) [70]. Therefore, we hypothesize the hands-on experience of BIBIFI may support secure development improvement in ways that MOOC lectures, without direct experience, do not.

5.7 Breaking Success

Now, we turn our attention to break-it team performance, i.e., how effective teams were at finding defects in build-it teams' submissions. First, we consider how and why teams performed as indicated by their (normalized) break-it score *prior to the fix-it phase*. We do this to measure a team's raw output, ignoring whether other teams found the same bug (which we cannot assess with confidence due to the lack of fix-it phase participation per Section 5.5). This data set includes 141 teams that participated in the break-it phase for the Secure Log, ATM, and Multiuser DB contests. We also model which factors contributed to *security bug count*, or how many total security bugs a break-it team found. Doing this disregards a break-it team's effort at finding correctness bugs.

We model both break-it score and security bug count using several of the same potential factors as discussed previously, but applied to the breaking team rather than the building team. In particular, we include the *Contest* they participated in, whether they were *MOOC* participants, the number of break-it *Team members*, average team-member *Coding experience*, average team-member *Knowledge of C*, and unique *Languages known* by the break-it team members. We also add two new potential factors. (1) Whether the breaking team also qualified as a *Build participant*.

Table 9. Factors and Baselines for Break-it Models

Factor	Description	Baseline
Contest	Secure Log, ATM, or Multiuser DB contest.	Secure Log
# Team members	A team's size.	—
Knowledge of C	The fraction of team members who know C or C++.	—
# Languages known	Number of programming languages team members know.	—
Coding experience	Average years of programming experience.	—
MOOC	If the team was participating in the MOOC capstone.	Non-MOOC
Build participant	If the breaking team qualified as a build participant.	Non-builder
Advanced techniques	If the breaking team used software analysis or fuzzing.	Not advanced

Table 10. Final Linear Regression Model of Teams' Break-it Scores, Indicating How Many Points Each Selected Factor Adds to the Total Score

Factor	Coef.	CI	p-value
Secure Log	—	—	—
ATM	**−2401.047**	**[−3781.59, −1020.5]**	**<0.001***
Multiuser DB	−61.25	[−1581.61, 1459.11]	0.937
# Team members	**386.975**	**[45.48, 728.47]**	**0.028***
Coding experience	87.591	[−1.73, 176.91]	0.057
Build participant	1260.199	[−315.62, 2836.02]	0.119
Knowledge of C	−1358.488	[−3151.99, 435.02]	0.14

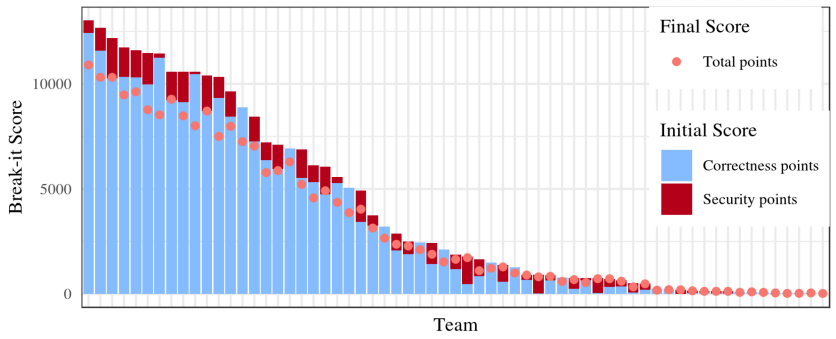
$R^2 = 0.15$.

(2) Whether the breaking team reported using *Advanced techniques* like software analysis or fuzzing to aid in bug finding. Teams that only used manual inspection and testing are not categorized as advanced. Thirty-four break-it teams (24%) reported using advanced techniques. These factors are summarized in Table 9.

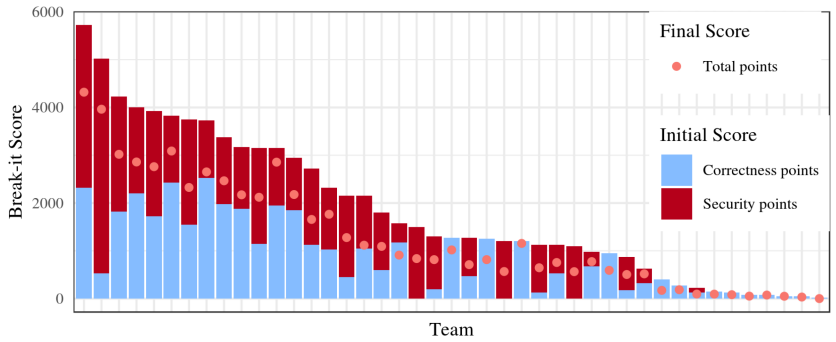
When carrying out the power analysis for these two models, we aimed for a *medium* effect size by Cohen's heuristic [25]. Assuming a power of 0.75, our conference paper considered a population of $N = 108$ for the Secure Log and ATM contests; with the eight degrees of freedom it yields a prospective effect size $f^2 = 0.136$. Including the Multiuser DB contest increases the degrees of freedom to nine and raises the population to $N = 141$. This yields a prospective effect size $f^2 = 0.107$, which (again) is an improvement over the initial analysis.

Break score. The model considering break-it score is given in Table 10. It has a coefficient of determination $R^2 = 0.15$, which is adequate. The model shows that teams with more members performed better, with an average of 387 additional points per team member. Auditing code for errors is an easily parallelized task, so teams with more members could divide their effort and achieve better coverage.

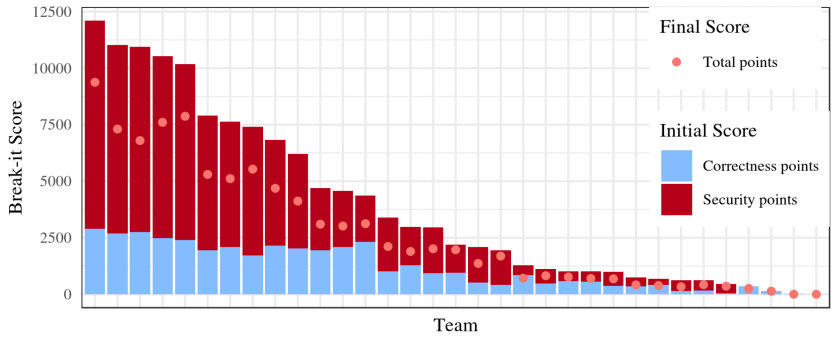
The model also indicates that Secure Log teams performed significantly better than ATM teams, and Multiuser DB teams performed similarly to ATM teams. Figure 8 illustrates that correctness bugs, despite being worth fewer points than security bugs, dominate overall break-it scores for the Secure Log contest. In the ATM contest, the scores are more evenly distributed between correctness and security bugs. This outcome is not surprising to us, as it was somewhat by design. The Secure Log problem defines a rich command-line interface with many opportunities for subtle correctness errors that break-it teams can target. It also allowed a break-it team to submit up to 10 correctness bugs per build-it submission. To nudge teams toward finding more security-relevant bugs, we



(a) Secure Log



(b) ATM



(c) Multiuser DB

Fig. 8. Scores of break-it teams prior to the fix-it phase, broken down by points from security and correctness bugs. The final score of the break-it team (after fix-it phase) is noted as a dot. Note the different ranges in the y-axes. In general, the Secure Log contest had the least proportion of points coming from security breaks.

reduced the submission limit from 10 to 5, and designed the ATM and Multiuser DB interface to be far simpler. For the Multiuser DB contest, an even greater portion of break-it scores come from security bugs. This again was by design as we increased the security bug limit. Instead of submitting a maximum of two security bugs against a specific build-it team, breakers could submit up to five security (or correctness) bugs against a given team.

Interestingly, making use of advanced analysis techniques did not factor into the final model; i.e., such techniques did not provide a meaningful advantage in our context. This makes sense when we

Table 11. Final Linear Regression Modeling the Count of Security Bugs Found by Each Team

Factor	Coef.	CI	p-value
Secure Log	—	—	—
Multiuser DB	**9.617**	**[5.84, 13.39]**	**<0.001***
ATM	**3.736**	**[0.3, 7.18]**	**0.035***
# Team members	**1.196**	**[0.35, 2.04]**	**0.006***
Build participant	**4.026**	**[0.13, 7.92]**	**0.045***

Coefficients indicate how many security bugs each factor adds to the count. $R^2 = 0.203$.

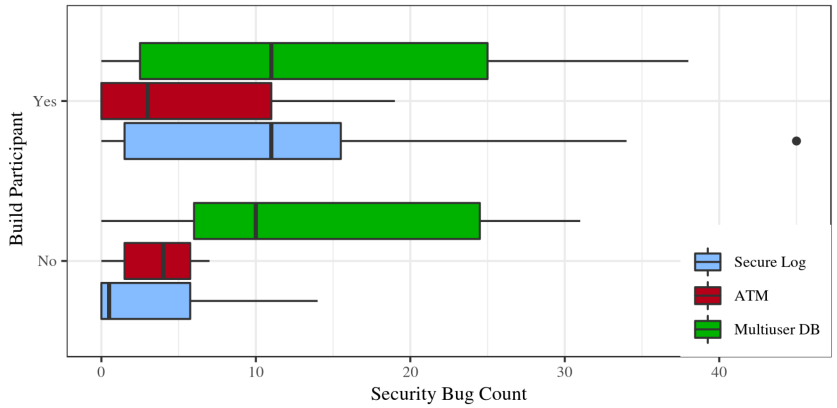


Fig. 9. Count of security bugs found by each break-it team, organized by contest and whether the team also participated in build-it. The heavy vertical line in the box is the median, the boxes show the first and third quartiles, and the whiskers extend to the most outlying data within ±1.5× the interquartile range. Dots indicate further outliers.

consider that such techniques tend to find generic errors such as crashes, bounds violations, or null pointer dereferences. Security violations for our problems are more often semantic, e.g., involving incorrect design or improper use of cryptography. Many correctness bugs were non-generic too, e.g., involving incorrect argument processing or mishandling of inconsistent or incorrect inputs.

Being a build participant and having more coding experience is identified as a positive factor in the break-it score, according to the model, but neither is statistically significant (though they are close to the threshold). Interestingly, knowledge of C is identified as a strongly negative factor in break-it score (though again, not statistically significant). Looking closely at the results, we find that *lack* of C knowledge is extremely *uncommon*, but that the handful of teams in this category did unusually well. However, there are too few of them for the result to be significant.

Again, we note MOOC participation was not included in our final model, suggesting this security education had at most a limited effect on breaking success.

Security bugs found. We next consider breaking success as measured by the count of security bugs a breaking team found. This model (Table 11) explains 20% of variance ($R^2 = 0.203$). The model again shows that team size is important, with an average of one extra security bug found for each additional team member. Being a qualified builder also significantly helps one's score; this makes intuitive sense, as one would expect to gain a great deal of insight into how a system could fail after successfully building a similar system. Figure 9 shows the distribution of the number

of security bugs found, per contest, for break-it teams that were and were not qualified build-it teams. Note that all but eight of the 141 break-it teams made some attempt, as defined by having made a commit, to participate during the build-it phase—most of these (115) qualified, but 18 did not. If the reason was that these teams were less capable programmers, then that may imply that programming ability generally has some correlation with break-it success.

On average, four more security bugs were found by ATM teams than Secure Log teams. This contrasts with the finding that Secure Log teams had higher overall break-it scores, but corresponds to the finding that more ATM submissions had security bugs found against them. As discussed above, this is because correctness bugs dominated the Secure Log contest but were not as dominant in the ATM contest. Once again, the reasons may have been the smaller budget on per-submission correctness bugs for the ATM contest, and the greater potential attack surface in the ATM problem.

Multiuser DB teams found ten more security bugs on average than Secure Log teams. One possible reason is that the Multiuser DB contest permitted teams to submit up to five security bug reports per target, rather than just two. Another is that with Multiuser DB it was easier for breakers to reuse break tests to see when multiple targets were susceptible to the same bug.

5.8 Model Differences

In the conference version of this article [56], we presented previous versions of these models with only data from the Secure Log and ATM contests. The updated models with Multiuser DB data are very similar to the original models, but there are some differences. We describe the differences to each model in this subsection.

Ship scores. In the original ship score model, students of the MOOC capstone performed 119 points better than non-MOOC teams. This correlation goes away when the Multiuser DB data is included. We hypothesize that this is due to prior coursework. MOOC students took three prior security courses that cover cryptography, which is directly relevant to the Secure Log and ATM problem, but not the Multiuser DB problem.

Lines of code is not statistically significant in the updated model, but it was significant in the original model. Each additional line of code corresponded with a drop of 0.03 points in ship score. Code size may not have improved ship scores for the Multiuser DB contest due to the nature of the problem. Teams needed to implement custom access control policies and there are fewer libraries available that implement this functionality.

Presence of security bugs. In the original presence-of-security-bugs model, lines of code was a significant factor. Each additional line of code slightly increased the likelihood of a security bug being present (1.001×). Lines of code is not in the latest model, which is similar to the change in the ship score model (Section 5.4). We hypothesize this change occured for the same reasons.

Break score. The break score model basically remained the same between versions. The only difference is the coefficients slightly changed with the addition of the Multiuser DB contest.

Security bugs found. The linear regression model for the number of security bugs found essentially remained unchanged. The only material change was the addition of the factor that found that Multiuser DB breakers found more security bugs than Secure Log problem breakers.

5.9 Summary

The results of our quantitative analysis provide insights into how different factors correlate with success in building and breaking software. Programs written in C and C++ received higher ship scores due to better performance. C/C++ submissions were also 11× more likely to have a reported security flaw than submissions written in statically typed languages.

Break-it teams with more team members found more security bugs and received more break-it points. Searching for vulnerabilities is easily parallelizable, so teams with more members could split the workload and audit more code. Successful build participants found more security bugs. This is intuitive as successfully building a program gives insights into the mistakes other similar programs might make.

6 RELATED WORK

BIBIFI bears similarity to existing programming and security contests but is unique in its focus on building secure systems. BIBIFI also is related to studies of code and secure development but differs in its open-ended contest format.

Contests. Cybersecurity contests typically focus on vulnerability discovery and exploitation, and sometimes involve system administration for defense. One popular style of contest, dubbed *capture the flag* (CTF), is exemplified by a contest held at DEFCON [4]. Here, teams run an identical system that has buggy components. The goal is to find and exploit the bugs in other competitors' systems while mitigating the bugs in your own. Compromising a system enables a team to acquire the system's key and thus "capture the flag." In addition to DEFCON CTF, there are other CTFs such as iCTF [24, 29, 65], S3 [16], KotH [20], and PicoCTF [22]. The use of this style of contest in an educational setting has been explored in prior work [28, 31, 38]. The Collegiate Cyber Defense Challenge [26, 27, 46] and the Maryland Cyber Challenge & Competition [8] have contestants defend a system, so their responsibilities end at the identification and mitigation of vulnerabilities. These contests focus on bugs in systems as a key factor of play, but neglect software development.

Since BIBIFI's inception, additional contests have been developed in its style. Make it and Break it [76] is an evaluation of the Build-it, Break-it, Fix-it type of contest. Two teams were tasked with building a secure internet of things (IoT) smart home with functionality including remote control of locks, speakers, and lighting. Teams then broke each other's implementations and found vulnerabilities like SQL injection and unauthorized control of locks. The contest organizers found this style of contest was beneficial in the development of cybersecurity skills and plan to run additional contests in the future. Git-based CTF [73] is similar to BIBIFI in that students were asked to implement a program according to a given specification. It differs in the fact that the CTF was fully run on Github and contestants used issue-tracking to submit breaks. In addition, builders were encouraged to fix breaks as soon as breaks were submitted, since they periodically lost points for unfixed breaks. This seems to have been an effective motivation for convincing builders to fix their mistakes, and it may be a solution to improve BIBIFI's fix-it participation.

Programming contests challenge students to build clever, efficient software, usually with constraints and while under (extreme) time pressure. The ACM programming contest [1] asks teams to write several programs in C/C++ or Java during a 5-hour time period. Google Code Jam [6] sets tasks that must be solved in minutes, which are then graded according to development speed (and implicitly, correctness). Topcoder [9] runs several contests; the Algorithm competitions are small projects that take a few hours to a week, whereas Design and Development competitions are for larger projects that must meet a broader specification. Code is judged for correctness (by passing tests), performance, and sometimes subjectively in terms of code quality or practicality of design. All of these resemble the build-it phase of BIBIFI but typically consider smaller tasks; they do not consider the security of the produced code.

Secure Development Practices and Advice. There is a growing literature of recommended practices for secure development. The BSIMM ("building security in" maturity model) [2] collects information from companies and places it within a taxonomy. Microsoft's Security Development Lifecycle (SDL) [40] describes possible strategies for incorporating security concerns into the development

process. Several authors make recommendations about development lifecycle and coding practices [19, 23, 39, 43, 60, 68]. Acar et al. collect and categorize 19 such resources [15].

Studies of secure software development. Researchers have considered how to include security in the development process. Work by Finifter and Wagner [35] and Prechelt [54] relates to both our build-it and break-it phases: they asked different teams to develop the same web application using different frameworks, and then subjected each implementation to automated (black box) testing and manual review. They found that both forms of review were effective in different ways, and that framework support for mitigating certain vulnerabilities improved overall security. Other studies focused on the effectiveness of vulnerability discovery techniques, e.g., as might be used during our break-it phase. Edmundson et al. [32] considered manual code review; Scandariato et al. [59] compared different vulnerability detection tools; other studies looked at software properties that might co-occur with security problems [37, 71, 77]. BIBIFI differs from all of these in its open-ended, contest format: Participants can employ any technique they like, and with a large enough population and/or measurable impact, the effectiveness of a given technique will be evident in final outcomes.

Other researchers have examined what factors influence the development of security errors. Common findings include developers who do not understand the threat model, security APIs with confusing options and poorly chosen defaults, and "temporary" test configurations that were not corrected prior to deployment [33, 34, 36]. Interview studies with developers suggest that security is often perceived as someone else's responsibility, not useful for career advancement, and not part of the "standard" developer mindset [47, 75]. Anecdotal recommendations resulting from these interviews include mandating and rewarding secure coding practices, ensuring that secure tools and APIs are more attractive than less secure ones, enable "security champions" with broadly defined roles, and favoring ongoing dialogue over checklists [17, 72, 74]. Developer Observatory [13, 14, 62] is an online platform that enables large-scale controlled security experiments by asking software developers to complete a security relevant programming tasks in the browser. Using this platform, Acar et al. studied how developer experience and API design for cryptographic libraries impact software security. Oliveira et al. [49] performed an experiment on security blindspots, which they define as a misconception, misunderstanding, or oversight by the developer in the use of an API. Their results indicate that API blindspots reduce a developer's ability to identity security concerns, I/O functionality is likely to cause blindspots, and experience does not influence a developer's ability to identify blindspots. Thompson [64] analyzed thousands of open source repositories and found that code review of pull requests reduced the number of reported security bugs.

Crash de-duplication. For accurate scoring, BIBIFI identifies duplicate bug reports by unifying the ones addressed by the same (atomic) fix. But this approach is manual, and relies on imperfect incentives. Other works have attempted to automatically de-duplicate bug reports, notably those involving crashes. Stack hashing [44] and AFL [11] coverage profiles offer potential solutions, however Klees et al. [42] show that fuzzers are poor at identifying which underlying bugs cause crashing inputs. Semantic crash bucketing [67] and symbolic analysis [52] show better results at mapping crashing inputs to unique bugs by taking into account semantic information of the program. The former supports BIBIFI's view that program fixes correspond to unique bugs.

7 CONCLUSION

This article has presented Build-it, Break-it, Fix-it (BIBIFI), a new security contest that brings together features from typical security contests, which focus on vulnerability detection and mitigation but not secure development, and programming contests, which focus on development but not security. During the first phase of the contest, teams construct software they intend to be

correct, efficient, and secure. During the second phase, break-it teams report security vulnerabilities and other defects in submitted software. In the final, fix-it, phase, builders fix reported bugs and thereby identify redundant defect reports. Final scores, following an incentives-conscious scoring system, reward the best builders and breakers.

During 2015 and 2016, we ran three contests involving a total of 156 teams and three different programming problems. Quantitative analysis from these contests found that the best performing build-it submissions used C/C++, but submissions coded in a statically typed language were less likely to have a security flaw. Break-it teams that were also successful build-it teams were significantly better at finding security bugs. Break-it teams with more members were more successful at breaking, since auditing code is a task that is easily subdivided.

The BIBIFI contest administration code is available at https://github.com/plum-umd/bibifi-code; data from our contests is available in limited form, upon request. More information, data, and opportunities to participate are available at https://builditbreakit.org.

ACKNOWLEDGMENTS

We thank Jandelyn Plane and Atif Memon who contributed to the initial development of BIBIFI and its preliminary data analysis. Many people in the security community, too numerous to list, contributed ideas and thoughts about BIBIFI during its development—thank you! Matthew Hou assisted with the qualitative analysis of contestant submissions. Manuel Benz, Martin Mory, Luke Valenta, Matt Blaze, Philip Ritchey, Aymeric Fromherz, Lujo Bauer, and Bryan Parno ran the contest at their universities and tested the infrastructure. Bobby Bhattacharjee and the anonymous reviewers provided helpful comments on drafts of this article.

REFERENCES

[1] ICPC Foundation. 2018. ACM-ICPC International Collegiate Programming Contest. Retrieved from http://icpc.baylor.edu.
[2] BSIMM. 2020. Building Security In Maturity Model (BSIMM). Retrieved from http://bsimm.com.
[3] DEF CON Communications. 2018. Capture the Flag Archive. Retrieved from https://www.defcon.org/html/links/dc-ctf.html.
[4] DEF CON Communications. 2010. DEF CON Hacking Conference. Retrieved from http://www.defcon.org.
[5] Git. 2020. Git – distributed version control management system. Retrieved from http://git-scm.com.
[6] Google. 2020. Google Code Jam. Retrieved from http://code.google.com/codejam.
[7] ICFP Programming Contest. 2019. Retrieved from http://icfpcontest.org.
[8] Federal Business Council. 2012. Maryland Cyber Challenge & Competition. Retrieved from http://www.fbcinc.com/e/cybermdconference/competitorinfo.aspx.
[9] TOPCODER. 2020. Top Coder competitions. Retrieved from http://apps.topcoder.com/wiki/display/tc/Algorithm+Overview.
[10] Michael Snoyman. 2020. Yesod Web Framework for Haskell. Retrieved from http://www.yesodweb.com.
[11] American Fuzzing Lop (AFL). 2018. Retrieved from http://lcamtuf.coredump.cx/afl/.
[12] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Info. Syst. Secur.* 13, 1 (2009), 4:1–4:40.
[13] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017. Comparing the usability of cryptographic APIs. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*.
[14] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. 2017. Security developer studies with GitHub users: Exploring a convenience sample. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS'17)*.
[15] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl. [n.d.]. Developers need support too: A survey of security advice for software developers. In *Proceedings of the IEEE Secure Development Conference (SecDev'17)*.
[16] Daniele Antonioli, Hamid Reza Ghaeini, Sridhar Adepu, Martin Ochoa, and Nils Ole Tippenhauer. 2017. Gamifying ICS security training and research: Design, implementation, and results of S3. In *Proceedings of the International Conference on Cyber-Physical Systems (CPS'17)*.

[17] Ingolf Becker, Simon Parkin, and M. Angela Sasse. 2017. Finding security champions in blends of security culture. In *Proceedings of the 2nd European Workshop on Usable Security (EuroUSEC'17)*. Internet Society.
[18] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *Proceedings of the International Conference on Cryptology and Information Security in Latin America*. Springer, 159–176.
[19] Paul E. Black, Lee Badger, Barbara Guttman, and Elizabeth Fong. 2016. *Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy*. Technical Report Draft NISTIR 8151. National Institute of Standards and Technology. Retrieved from http://csrc.nist.gov/publications/drafts/nistir-8151/nistir8151_draft.pdf.
[20] Kevin Bock, George Hughey, and Dave Levin. 2018. King of the hill: A novel cybersecurity competition for teaching penetration testing. In *Proceedings of the USENIX Workshop on Advances in Security Education (ASE'18)*.
[21] Kenneth P. Burnham, David R. Anderson, and Kathryn P. Huyvaert. 2011. AIC model selection and multimodel inference in behavioral ecology: Some background, observations, and comparisons. *Behav. Ecol. Sociobiol.* 65, 1 (2011), 23–35.
[22] Peter Chapman, Jonathan Burket, and David Brumley. 2014. PicoCTF: A game-based computer security competition for high school students. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE'14)*.
[23] Brian Chess and Jacob West. 2007. *Secure Programming with Static Analysis*. Addison-Wesley.
[24] Nicholas Childers, Bryce Boe, Lorenzo Cavallaro, Ludovico Cavedon, Marco Cova, Manuel Egele, and Giovanni Vigna. 2010. Organizing large scale hacking competitions. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*.
[25] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates.
[26] Art Conklin. 2005. The use of a collegiate cyber defense competition in information security education. In *Proceedings of the Information Security Curriculum Development Conference (InfoSecCD'05)*.
[27] Art Conklin. 2006. Cyber defense competitions and information security education: An active learning solution for a capstone course. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS'06)*.
[28] Gregory Conti, Thomas Babbitt, and John Nelson. 2011. Hacking competitions and their untapped potential for security education. *Secur. Privacy* 9, 3 (2011), 56–59.
[29] Adam Doupé, Manuel Egele, Benjamin Caillat, Gianluca Stringhini, Gorkem Yakin, Ali Zand, Ludovico Cavedon, and Giovanni Vigna. 2011. Hit'Em where it hurts: A live security exercise on cyber situational awareness. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'11)*.
[30] dragostech.com inc. [n.d.]. CanSecWest Applied Security Conference. Retrieved from http://cansecwest.com.
[31] Chris Eagle. 2013. Computer security competitions: Expanding educational outcomes. *Secur. Privacy* 11, 4 (2013).
[32] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. 2013. An empirical study on the effectiveness of security code review. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSOS'13)*.
[33] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM, 73–84.
[34] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL development in an appified world. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*.
[35] Matthew Finifter and David Wagner. 2011. Exploring the relationship betweenweb application development tools and security. In *Proceedings of the USENIX Conference on Web Application Development (WebApps'11)*.
[36] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'12)*. ACM.
[37] Keith Harrison and Gregory White. 2010. An empirical study on the effectiveness of common security measures. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS'10)*.
[38] Lance J. Hoffman, Tim Rosenberg, and Ronald Dodge. 2005. Exploring a national cybersecurity exercise for universities. *Secur. Privacy* 3, 5 (2005), 27–33.
[39] Michael Howard and David LeBlanc. 2003. *Writing Secure Code*. Microsoft Press.
[40] Michael Howard and Steve Lipner. 2006. *The Security Development Lifecycle*. Microsoft Press.
[41] Queena Kim. 2014. Want to learn cybersecurity? Head to Def Con. Retrieved from http://www.marketplace.org/2014/08/25/tech/want-learn-cybersecurity-head-def-con.
[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*.
[43] Gary McGraw. 2006. *Software Security: Building Security In*. Addison-Wesley.

[44] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the USENIX Security Symposium*.

[45] N. J. D. Nagelkerke. 1991. A note on a general definition of the coefficient of determination. *Biometrika* 78, 3 (09 1991), 691–692.

[46] National Collegiate Cyber Defense Competition. [n.d.]. Retrieved from http://www.nationalccdc.org.

[47] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'14)*.

[48] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. 2018. API blindspots: Why experienced developers write vulnerable code. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS'18)*.

[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. 2018. API blindspots: Why experienced developers write vulnerable code. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS'18)*.

[50] OWASP. 2010. Secure Coding Practices - Quick Reference Guide. Retrieved from https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf.

[51] James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information flow security for multi-tier web applications. *Proc. ACM Program. Lang.* 3 (Jan. 2019).

[52] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing failing tests via symbolic analysis. In *Proceedings of the International Conference on Fundeamental Approaches to Software Engineering (FASE'17)*.

[53] Polytechnic Institute of New York University. [n.d.]. CSAW—CyberSecurity Competition 2012. Retrieved from http://www.poly.edu/csaw2012/csaw-CTF.

[54] L. Prechelt. 2011. Plat_Forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Trans. Softw. Eng.* 37, 1 (2011), 95–108.

[55] psql [n.d.]. PostgreSQL: The world's most advanced open source database. Retrieved from http://www.postgresql.org.

[56] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. 2016. Build it, break it, fix it: Contesting secure development. In *Proceedings of the Conference on Computer and Communications Security (CCS'16)*.

[57] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Atif Memon, Jandelyn Plane, and Piotr Mardziel. 2015. Build it break it: Measuring and comparing development security. In *Proceedings of the International Conference on Cyber Security for Emerging Technologies (CSET'15)*.

[58] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[59] Riccardo Scandariato, James Walden, and Wouter Joosen. 2013. Static analysis versus penetration testing: A controlled experiment. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'13)*.

[60] Robert C. Seacord. 2013. *Secure Coding in C and C++*. Addison-Wesley.

[61] Deian Stefan, Alejandro Russo, John Mitchell, and David Mazieres. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the ACM SIGPLAN Haskell Symposium*.

[62] Christian Stransky, Yasemin Acar, Duc Cuong Nguyen, Dominik Wermke, Doowon Kim, Elissa M. Redmiles, Michael Backes, Simson Garfinkel, Michelle L. Mazurek, and Sascha Fahl. 2017. Lessons learned from using an online platform to conduct large-scale, online controlled security experiments with software developers. In *Proceedings of the International Conference on Cyber Security for Emerging Technologies (CSET'17)*.

[63] Positive Technologies. 2018. ATM logic attacks: scenarios, 2018. Retrieved from https://www.ptsecurity.com/upload/corporate/ww-en/analytics/ATM-Vulnerabilities-2018-eng.pdf.

[64] Christopher Thompson and David Wagner. 2017. A large-scale study of modern code review and security in open source projects. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'17)*.

[65] Erik Trickel, Francesco Disperati, Eric Gustafson, Faezeh Kalantari, Mike Mabey, Naveen Tiwari, Yeganeh Safaei, Adam Doupé, and Giovanni Vigna. 2017. Shell we play a game? CTF-as-a-service for security education. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*.

[66] Úlfar Erlingsson. 2012. *personal communication stating that CFI was not deployed at Microsoft due to its overhead exceeding 10%.*

[67] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'18)*.

[68] John Viega and Gary McGraw. 2001. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley.

[69] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *Proceedings of the 29th USENIX Security Symposium (USENIXSecurity'20)*. USENIX Association.

[70] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. 2018. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *Proceedings of the IEEE IEEE Symposium on Security and Privacy (S&P'18)*.

[71] James Walden, Jeff Stuckman, and Riccardo Scandariato. 2014. Predicting vulnerable components: Software metrics vs text mining. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*.

[72] Charles Weir, Awais Rashid, and James Noble. 2017. I'd like to have an argument, please: Using dialectic for effective app security. In *Proceedings of the 2nd European Workshop on Usable Security (EuroUSEC'17)*. Internet Society.

[73] SeongIl Wi, Jaeseung Choi, and Sang Kil Cha. 2018. Git-based CTF: A simple and effective approach to organizing in-course attack-and-defense security competition. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*.

[74] Glenn Wurster and P. C. van Oorschot. 2008. The developer is the enemy. In *Proceedings of the New Security Paradigms Workshop (NSPW'08)*. 89.

[75] J. Xie, H. R. Lipford, and B. Chu. 2011. Why do programmers make security errors? In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6070393.

[76] Muhammad Mudassar Yamin, Basel Katt, Espen Torseth, Vasileios Gkioulos, and Stewart James Kowalski. 2018. Make it and break it: An IoT smart home testbed case study. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*.

[77] Joonseok Yang, Duksan Ryu, and Jongmoon Baik. 2016. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *Proceedings of the International Conference on Big Data and Smart Computing (BigComp'16)*.

Received June 2019; revised December 2019; accepted February 2020