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

Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel
University of Maryland

ABSTRACT

Typical security contests focus on breaking or mitigating the impact of buggy systems. We present the Build-it, Break-it, Fix-it (BIBIFI) contest, which aims to assess the ability to securely build software, not just break it. In particular, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. We analyzed data gathered from contest runs involving two distinct programming problems attempted by 116 teams and discovered several factors that statistically correlate with good performance.

1. INTRODUCTION

Capture-the-flag (CTF) and other cybersecurity contests [15, 16, 5, 18, 6] are popular and serve as valuable proving grounds for finding cybersecurity talent. These contests largely focus on *breaking* (e.g., exploiting vulnerabilities and/or misconfigurations) and *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 [25, 2, 12] 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 [22], not something that can be added after the fact. As such, we should not assume that good breakers will necessarily be good builders [14], or that top coders can produce secure systems.

This paper presents **Build-it, Break-it, Fix-it** (BIBIFI), a new security contest with a focus on *building secure systems*. A BIBIFI contest has three phases. The first phase, *Build-it*, asks small development teams to build software according to a provided specification 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, benefit a break-it team's score and penalize the build-it team's score; more points are assigned to security-relevant problems. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, asks builders to fix bugs and thereby get points back if the process discovers that distinct break-it test cases identify the same defect. BIBIFI's structure and scoring system are carefully designed to encourage meaningful outcomes, e.g., to ensure that the top-scoring build-it teams really produce secure and efficient software. Behaviors that would thwart such outcomes are discouraged. For example, break-it teams may submit a limited number of bug reports per build-it submission, and will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to 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, etc. can influence a team's (non)success in the build-it and/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 paper studies the outcomes of three BIBIFI contests that we held during 2015, involving two 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 and third contests were run simultaneously. They 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) or corrupt it (e.g., stealing or adding money to accounts). Two of the three contests drew participants from a MOOC (Massive Online Open Courseware) course on cybersecurity. These participants (278 total, comprising 109 teams) each had an average of 10 years of programming experience and had just completed a four-course sequence including courses on secure software and cryptography. The third contest involved

U.S.-based graduate and undergraduate students (23 total, comprising 6 teams) with less experience and training.

Rigorous quantitative analysis of the outcomes 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 found later. Teams with broader programming language knowledge and that wrote less code also produced more secure implementations. Considering break-it scores: larger teams found more bugs during the break-it phase. Greater programming experience, and knowledge of C, were also helpful. Break-it teams that also participated during the build-it phase were significantly more likely to find a security bug than those that did not.

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 freed up worry of proper use of randomness, nonces, etc. Unsuccessful teams typically failed to employ cryptography, implemented it incorrectly, used insufficient randomness, or failed to use authentication; there were comparatively few memory safety violations. Break-it teams found clever ways to exploit security problems; some MITM implementations were quite sophisticated.

In summary, this paper makes two main contributions. First, it presents BIBIFI, a new security contest that encourages building, not just breaking. Second, it presents a detailed description of three BIBIFI contests along with both a quantitative and qualitative analysis of the results. We will be making the BIBIFI code and infrastructure publicly available so that others may run their own competitions; we hope that this opens up a line of research built on empirical experiments with secure programming methodologies.[1]

The rest of this paper is organized as follows. We present the design of BIBIFI in §2 and describe specifics of the contests we ran in §3. We present the quantitative analysis of the data we collected from these contests in §4, and qualitative analysis in §5. We review related work in §6 and conclude in §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).

2.1 Competition phases

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

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 administrators. 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 [9]; with each push, the contest infrastructure downloads the submission, builds it, tests it (for correctness and performance), and updates the scoreboard. §3 describes the two target problems we developed: (1) an append-only log; and (2) a pair of communicating programs that simulate a bank and ATM.

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. We randomize each break-it team's view of the build-it teams' submissions,[2] but organize them by meta-data like programming language. When they think they have found a defect, breakers submit a test case that exposes the defect and an explanation of the issue. To encourage coverage, a break-it team may only submit up a fixed number of test cases per build-it submission. 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.[3] More points are awarded to clear security problems, which may be demonstrated using alternative test formats. The auto-judgment approaches we developed for the two different contest problems are described in §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 best builders and best breakers as determined by the scoring system described next.

2.2 Competition scoring

[1] This paper subsumes a previously published 6-page workshop paper [21]. The initial BIBIFI design and implementation also appeared in that paper, as did a brief description of a pilot run of the contest. This paper presents many more details about the contest setup along with a quantitative and qualitative analysis of the outcomes of several larger contests.

[2] This avoids spurious unfair effects, such as if break-it teams investigating code in the order in which we give it to them.

[3] To encourage testing of our own infrastructure, teams can also earn points by finding bugs in the oracle.

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, creative analysis of others' code. The scoring rules 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, and efficient. While security is of primary interest to our contest, developers in practice must balance these other aspects of quality against security [1, 26], leading to a set of trade-offs that cannot be ignored if we wish to understand real developer decision-making.

To encourage these, each build-it team's score is the sum of the *ship* score[4] 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 $M \cdot (worst - v)/(worst - best)$, 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.

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 $2M$.

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 a particular 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.

After the break-it phase, a break-it team's score is the summed value of all defects they have found, using the above P valuations. After the fix-it phase, this score is reduced. In particular, 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 $4\times$ more points for security-relevant 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 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.

2.2.3 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. 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.[5] We observed this behavior for our contests, as described in §4.5. Both actions decrease a team's resilience score (and correspondingly increase breakers' scores). We can mitigate these issues with sufficiently strong incentives, e.g., by offering prizes to all participants commensurate with their final score, rather than offering prizes only to winners.

Finally, there are several design points in the problem definition 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 (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. 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 paper, we are fairly comfortable with these design points. In particular, our earlier contest [21] prioritized security bugs 2-to-1 and had fewer interesting performance tests, and outcomes were better when we increased the ratio.

2.2.4 Discouraging collusion

BIBIFI contestants may form teams however they wish, and may participate remotely. This encourages wider partic-

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

[5]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.

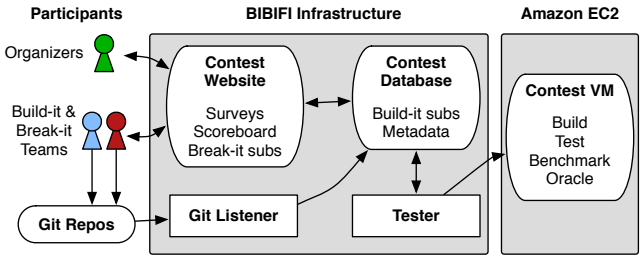


Figure 1: Overview of BIBIFI's implementation.

ipation, 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 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 and other 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 ∼11000 lines of Haskell using the Yesod [29] web framework backed by a PostgreSQL [20] 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 LMonad [17], which is derived from LIO [24], in order to protect against inadvertent information leaks and privilege escalations. LMonad dynamically guarantees that users can only access their own information.

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 ∼5100 lines of Haskell code (and a little Python).

To automate testing, we require contestants to specify a URL to a Git [9] repository hosted on either 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 an Amazon EC2 virtual machine which 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 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.

3. CONTEST PROBLEMS

This section presents the two programming problems we developed for the contests held during 2015, including problem-specific notions of security defect and how breaks exploiting such defects were tested.

3.1 Secure log (Spring 2015)

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 people), but they could implement two more for extra points (involving time spent in the museum, and intersections among different peoples' 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 deny the requests.

A canonical way of implementing the secure log is 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/MAC 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. Generally speaking, teams may also be tempted to implement their own encryption rather than use existing libraries, or to simply sidestep encryption altogether. §5 reports several cases we observed.

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

Integrity breaks. For about half of the generated log files we also provided the transcript of the `logappend` operations (*sans* auth token) 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. 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 Securing ATM interactions (Fall 2015)

The ATM problem asked builders to construct two communicating programs: `atm` acts as an ATM client, allowing customers to set up an account, and deposit and withdraw money, while `bank` is a server that processes client requests, tracking bank balances. `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.[6] 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.

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 the 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 the 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 §5.

Build-it submissions' performance was measured as the time to complete a series of benchmarks involving various `atm`/`bank` interactions.[7] Correctness (and *crash*) bug reports were defined as sequences of `atm` commands where the targeted submission produces different outputs than the oracle (or crashes). Security defects were 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,[8] and accepts a connection from the `atm` while connecting to the `bank`. The MITM program can thus observe and/or modify communications between `atm` and `bank`, as well as drop messages or initiate its own. We provided a Python-based proxy as a starter MITM: It sets up the connections and forwards communications between the two endpoints.

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), con-

[6] In a real deployment, this might be done by "burning" the auth file into the ATM's ROM prior to installing it.

[7] This transcript was always serial, so there was no direct motivation to support parallelism for higher throughput.

[8] All submissions were required to communicate via TCP.

sider a submission that uses deterministic encryption without nonces in messages. The MITM could direct the command server to withdraw money from an account, and then replay the message it observes. When run on the vulnerable submission, this would debit 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.

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), the break is considered successful. Before demonstrating knowledge of the secret, the MITM can send commands to the server with a *symbolic* "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.

As with the 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 against each submission.

Timeouts and denial of service. One difficulty with our use of a MITM is that we cannot reliably detect bugs in submissions 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 reporting any such errors, meaning that we may miss some; i.e., flaws in builder implementations might exist but evidence of those bugs might not be realizable in our testing system.

4. QUANTITATIVE ANALYSIS

This section analyzes data we have gathered from three contests we ran during 2015.[9] We consider participants' performance in each phase of the contest, including which factors contribute to high ship scores, resistance to breaking by other teams, and strong performance as breakers. We find that on average, teams that program in languages other than C and C++, and those whose members know more programming languages (perhaps a proxy for overall programming skill), are less likely to have security bugs identified in their code. 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 in the build-it phase (and therefore having given thought to how to secure an implementation). Overall, the Fall 2015 contest, which used the ATM problem, was associated with more security bugs than the Spring 2015 secure log contest.

4.1 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, etc. 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.

4.2 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. This allows us to estimate how each factor impacts the likelihood of an outcome.

We consider many variables that could potentially impact teams' results. To avoid over-fitting, we initially select as potential factors those variables that we believe are of most interest, within acceptable limits for power and effect size. (Our choices are detailed below.) In addition, we test models with all possible combinations of these initial factors and select the model with the minimum Akaike Information Criterion (AIC) [3]. Only the final models are presented.

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 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.

4.3 Contestants

We consider three contests offered at two times:

Spring 2015: We held one contest during May–June 2015 as the capstone to a Cybersecurity MOOC sequence.[10] Before completing in the capstone, participants passed courses on software security, cryptography, usable security, and hard-

[9] We also ran a contest during Fall 2014 but exclude it from consideration due to differences in how it was administered.

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

Contest	USA	India	Russia	Brazil	Other
Spring 2015	30	7	12	12	120
Fall 2015	64	14	12	20	110

Table 1: Contestants, by self-reported country.

Contest	Spring 15†	Fall 15†	Fall 15
# Contestants	156	122	23
% Male	91%	89%	100%
% Female	5%	9 %	0 %
Age (mean/min/max)	34.8/20/61	33.5/19/69	25.1/17/31
% with CS degrees	35 %	38 %	23 %
Years programming	9.6/0/30	9.9/0/37	6.6/2/13
# Build-it teams	61	34	6
Build-it team size	2.2/1/5	3.1/1/5	3.1/1/6
# Break-it teams (that also built)	65 (58)	39 (32)	4 (3)
Break-it team size	2.4/1/5	3.0/1/5	3.5/1/6
# PLs known per team	6.8/1/22	10.0/2/20	4.2/1/8

Table 2: Demographics of contestants from qualifying teams. † indicates MOOC participants. Some participants declined to specify gender.

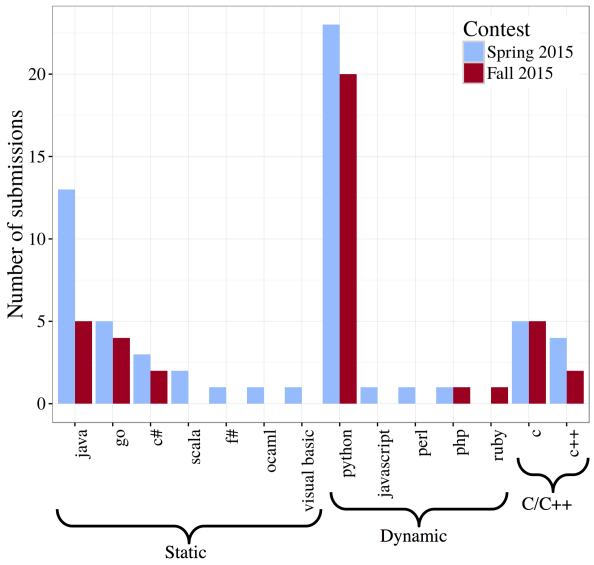


Figure 2: The number of build-it submissions in each contest, organized by primary programming language used. The brackets group the languages into categories.

ware security. The contest problem was the secure log problem (§3.1).

Fall 2015: During Oct.–Nov. 2015 we offered two contests simultaneously, one as a MOOC capstone, and the other open to U.S. college students. We merged the contests after the build-it phase, due to low participation in the open contest; from here on we refer to these two as a single contest. The contest problem was the ATM problem (§3.2).

The majority of all of our contestants came from the U.S. There was 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 1, and additional information about contestant demographics is presented in Table 2.

4.4 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 (§2.1). This data set contains all 101 teams from the Spring 2015 and Fall 2015 contests that qualified after the build-it phase. Both 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 both contests, which effectively normalizes the scores.

Model setup. To ensure enough power to find meaningful relationships, we decided to aim for a prospective effect size roughly equivalent to Cohen's *medium* effect heuristic, $f^2 = 0.15$ [4]. An effect this size suggests the model can explain up to 13% of the variance in the outcome variable. With an assumed power of 0.75 and population $N = 101$, we limited ourselves to nine degrees of freedom, which yields a prospective $f^2 = 0.154$. (Observed effect size for the final model is reported with the regression results below.) Within this limit, we selected the following potential factors:

Contest: Whether the team's submission was for the Spring 2015 contest or the Fall 2015 contest.

 # Team members: A team's size.

Knowledge of C: The fraction of team members who listed C or C++ as a programming language they know. We included this variable 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: How many unique programming languages team members collectively claim to know (see the last row of Table 2). 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.

Coding experience: The average years of programming experience reported by a team's members.

Lines of code: The SLOC[11] count of lines of code for the team's final submission at qualification time.

Language category: We manually identified each team's submission as having one "primary" language. These languages were then assigned to one of three categories: C/C++, statically-typed (e.g., Java, Go, but not C/C++) and dynamically-typed (e.g., Perl, Python). C/C++ is the baseline category for the regression. Precise category allocations, and total submissions for each language, segregated by contest, are given in Figure 2.

MOOC: True if the team was participating in the MOOC capstone project; otherwise false.

Results. Our regression results (Table 3) indicate that ship score is strongly correlated with language choice. Teams that programmed in C or C++ performed on average 121 and 92 points better than those who programmed in dynamically typed and statically typed languages, respectively.

[11] http://www.dwheeler.com/sloccount

Figure 3: Each team's ship score, compared to the lines of code in its implementation and organized by language category. Fewer LOC and using C/C++ correlate with a higher ship score.

Figure 3 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 Spring 2015 implemented all optional features, while six teams in the other categories implemented only 6 of 10 and one team implemented none; the Fall 2015 contest offered no optional features. We artificially increased the scores of those seven teams as if they had implemented all optional features and reran the regression model. The resulting model had very similar coefficients.

Our results also suggest that teams who were associated with the MOOC capstone performed 119 points better than non-MOOC teams. MOOC participants typically had more programming experience and CS training, so perhaps that is the reason.

Finally, we found that each additional line of code in a team's submission was associated with a drop of 0.03 points in ship score. Based on our qualitative observations (see §5), we hypothesize this may relate to more reuse of code from libraries, which are not counted in a team's LOC (most libraries were installed directly on the VM, not in the submission itself). As shown in Figure 3, LOC is also (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).

4.5 Code quality measures

Now we turn to measures of a build-it submission's quality—in terms of its correctness and security—based on how it held up under scrutiny by break-it teams.

Factor	Coef.	SE	p-value
Fall 2015	-21.462	28.359	0.451
Lines of code	-0.031	0.014	0.036*
Dynamically typed	-120.577	40.953	0.004*
Statically typed	-91.782	39.388	0.022*
MOOC	119.359	58.375	0.044*

Table 3: Final linear regression model of teams' ship scores, indicating how many points each selected factor adds to the total score. Overall effect size $f^2 = 0.163$.

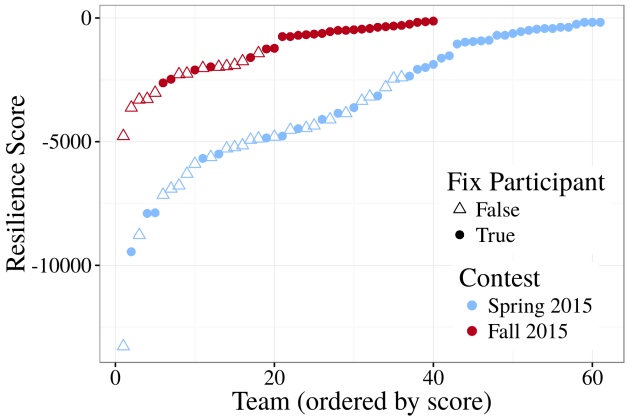


Figure 4: Final resilience scores, ordered by team, and plotted for each contest problem. Build-it teams who did not bother to fix bugs generally had lower scores.

Resilience. The total build-it score is the sum of ship score, just discussed, and *resilience*. Resilience is a non-positive score that derives from break-it teams' test cases that prove the presence of defects. Builders may increase this score during the fix-it phase, as fixes prevent double-counting test cases that identify the same defect (see §2.1).

Unfortunately, upon studying the data we found that a large percentage of build-it teams opted not to fix bugs reported against their code, forgoing the scoring advantage of doing so. We can see this in Figure 4, which graphs the resilience scores (Y-axis) of all teams, ordered by score, for the two 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. We further confirmed that fixing, or not, was a dominant factor by running a regression on resilience score that included fix-it phase participation as a factor (not shown).

This situation is disappointing, as we cannot treat resilience score as a good measure of code quality (when added to ship score). Our hypothesis is that participants were not sufficiently incentivized to fix bugs, for two reasons. First, teams that are 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 bugs was unnecessary for attaining this minimum score. We are exploring alternative structures that more strongly incentivize all teams to fix all (duplicated) bugs.

Factor	Coef.	Exp(coef)	SE	p-value
Fall 2015	5.692	296.395	1.374	<0.001*
# Languages known	-0.184	0.832	0.086	0.033*
Lines of code	0.001	1.001	0.0003	0.030*
Dynamically typed	-0.751	0.472	0.879	0.393
Statically typed	-2.138	0.118	0.889	0.016*
MOOC	2.872	17.674	1.672	0.086

Table 4: Final logistic regression model, measuring log likelihood of a security bug being found in a team's submission.

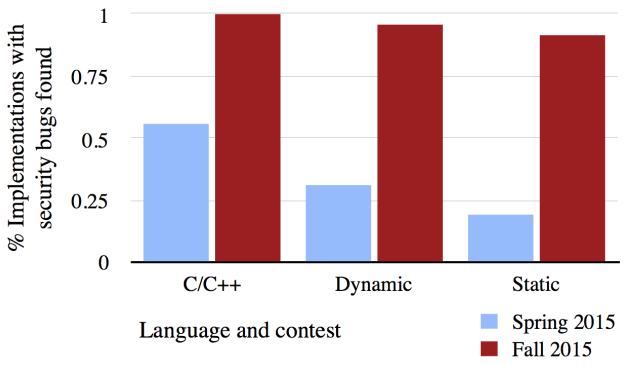


Figure 5: The fraction of teams in whose submission a security bug was found, for each contest and language category.

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, or integrity defect is demonstrated. In this model we used logistic regression over the same set of factors as the ship model.

Table 4 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 roughly how strongly that factor being true affects the likelihood relative to the baseline category.[12] For numeric factors, the exponential indicates how the likelihood changes with each unit change in that factor.

Fall 2015 implementations were 296× as likely as Spring 2015 implementations to have a security bug discovered against them.[13] We hypothesize this is due to the increased security design space in the ATM problem as compared to the gallery problem. Although it is easier to demonstrate a security error in the gallery 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 §5.

The model also shows that C/C++ implementations were more likely to contain an identified security bug than either static or dynamic implementations. For static languages, this effect is significant and indicates that a C/C++ program was about 8.5× (that is, 1/0.118) as likely to contain an identified bug. This effect is clear in Figure 5, which plots the fraction of implementations that contain a security bug, broken down by language type and contest problem. Note that the large majority of these were integrity/privacy bugs, not memory safety issues (crashes).

Our model shows that teams that knew more unique languages (even if they did not use those languages in their submission) performed slightly better, about 1.2× for each language known. Additional LOC in an implementation were also associated with a very small increase in the presence of an identified security bug.

Finally, the model shows two factors that played a role in the outcome, but not in a statistically significant way: using a dynamically typed language, and participating in the MOOC. We see the effect of the former in Figure 5. For the latter, the effect size is quite large; it's possible that the MOOC security training played a role.

[12] In cases (such as the Fall 2015 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.

[13] This coefficient is somewhat exaggerated (see prior footnote), but the difference between contests is large and significant.

4.6 Breaking success

Now we turn our attention to break-it team performance, i.e., how effective teams were at finding defects in others' 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 §4.5). This data set includes 108 teams that participated in the break-it phase in Spring and Fall 2015. 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 which 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:

Build participant: True if the breaking team participated in the build-it phase, otherwise false.

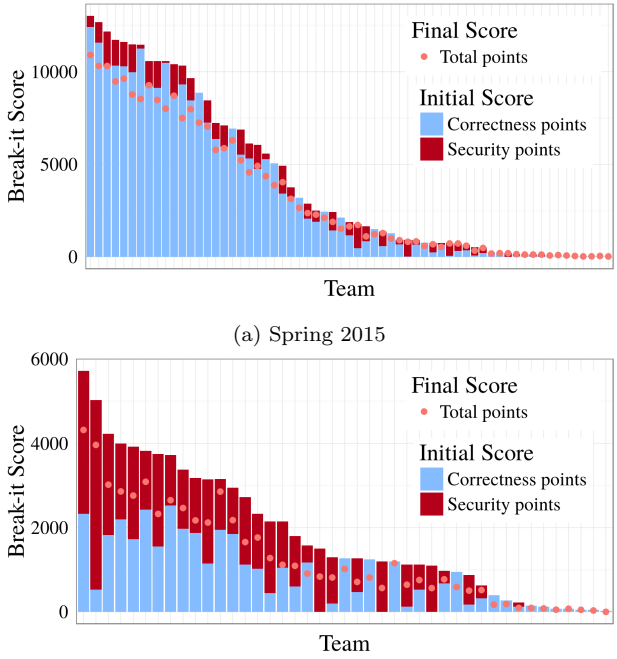
Advanced techniques: True if the breaking team reported using software analysis or fuzzing to aid in bug finding. Teams that only used manual inspection and testing are categorized as false.

For these two initial models, our potential factors provide eight degrees of freedom; again assuming power of 0.75, this yields a prospective effect size $f^2 = 0.136$, indicating we could again expect to find effects of roughly medium size by Cohen's heuristic [4].

Break score. The model considering break-it score is given in Table 5. It shows that teams with more members performed better, with an average of 430 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. Recall that having more

Factor	Coef.	SE	p-value
Fall 2015	-2406.89	685.73	<0.001*
# Team members	430.01	193.22	0.028*
Knowledge of C	-1591.02	1006.13	0.117
Coding experience	99.24	51.29	0.056
Build participant	1534.13	995.87	0.127

Table 5: Final linear regression model of teams' break-it scores, indicating how many points each selected factor adds to the total score. Overall effect size $f^2 = 0.039$.



(a) Spring 2015

(b) Fall 2015

Figure 6: 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 Spring 2015 contest (secure log problem) had higher scores for breaking.

team members did not help build-it teams (see Tables 3 and 4); this makes sense as development requires more coordination, especially during the early stages.

The model also indicates that Spring 2015 teams performed significantly better than Fall 2015 teams. Figure 6 illustrates that correctness bugs, despite being worth fewer points than security bugs, dominate overall break-it scores for Spring 2015. In Fall 2015 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 Spring 2015 problem defines a rich command-line interface with many opportunities for subtle 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 reduced the submission limit from 10 to 5, and designed the Fall 2015 interface to be far simpler.

Interestingly, making use of advanced analysis techniques

Factor	Coef.	SE	p-value
Fall 2015	3.847	1.486	0.011*
# Team members	1.218	0.417	0.004*
Build participant	5.430	2.116	0.012*

Table 6: Final linear regression modeling the count of security bugs found by each team. Coefficients indicate how many security bugs each factor adds to the count. Overall effect size $f^2 = 0.035$.

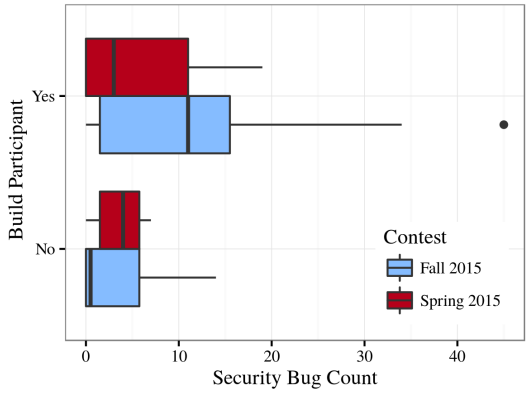


Figure 7: Count of security bugs found by each break-it team, organized by contest and whether the team also participated in build-it. The heavy vertical line in the box is the median, the boxes show the first and third quartiles, and the whiskers extend to the most outlying data within $\pm 1.5\times$ the interquartile range. Dots indicate further outliers.

did not factor into the final model.

Being a build participant and having more coding experience is identified as a postive factor in break-it score, according to the model, but neither is statistically significant. 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 to few of them for the result to be significant.

Security bugs found. We next consider breaking success as measured by the count of security bugs a breaking team found. This model (Table 6) again shows that team size is important, with an average of one extra security bug found for each additional team member. Being a 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 building a similar system. Figure 7 shows the distribution of the number of security bugs found, per contest, for break-it teams that were and were not build-it teams.

On average, four more security bugs were found by a Fall 2015 team than a Spring 2015 team. This contrasts with the finding that Spring 2015 teams had higher overall break-it scores, but corresponds to the finding that more Fall 2015 submissions had security bugs found against them. As dis-

cussed above, this is because correctness bugs dominated in Spring 2015 but were not as dominant in Fall 2015. Once again, the reasons may have been the smaller budget on per-submission correctness bugs in Fall 2015, and the greater potential attack surface in the ATM problem.

5. QUALITATIVE ANALYSIS

As part of the data gathered, we also have the entire program produced during the build-it phase as well as the programs patched during the fix-it phase. We can then perform a qualitative analysis of the programs which is guided by knowing the security outcome of a given program. Did lots of break-it teams find bugs in the program, or did they not? What are traits or characteristics of well-designed programs?

5.1 Success Stories

The success stories bear out some old chestnuts of wisdom in the security community: submissions that fared well through the break-it phase made heavy use of existing high-level cryptographic libraries with few "knobs" that allow for incorrect usage.

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. This put the contestant on good footing; to find bugs in this system, other contestants would need to break the security of OpenSSL.

Another implementation, also for the ATM problem, 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 first break the security of NaCl.

An implementation of the log reader problem, also written in Java, achieved success using a high level API. They used the BouncyCastle library to construct a valid encrypt-then-MAC scheme over the entire log file.

5.2 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 it is a similar dynamic observed in the software marketplace today.

Many implementations of the log problem lacked encryption or authentication. 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 which they could exploit with at most fifty probes. The target submission truncated the "authentication token," so that it was vulnerable to a brute force attack.

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 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.

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.

Some failures were common across log implementations as well: if an implementation used encryption, it might not use authentication. If it used authentication, it would authenticate records stored in the file individually and 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.

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 a system administration component for defense. One popular style of contest is dubbed *capture the flag* (CTF) and is exemplified by a contest held at DEFCON [13]. 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." The Collegiate Cyber Defense Challenge [16] and the Maryland Cyber Challenge & Competition [15] 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.

Programming contests challenge students to build clever, efficient software, usually with constraints and while under (extreme) time pressure. The ACM programming contest [2] asks teams to write several programs in C/C++ or Java during a 5-hour time period. Google Code Jam [10] sets tasks that must be solved in minutes, which are then graded according to development speed (and implicitly, correctness). Topcoder [25] 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.

Studies of secure software development. There have been a few studies of different methods and techniques for ensuring security. Work by Finifter and Wagner [8] and Prechelt [19] 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. [7] considered manual code review; Scandariato et al. [23] compared different vulnerability detection tools; other studies looked at software properties that might co-occur with security problems [27, 28, 11]. 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.

7. CONCLUSIONS

This paper 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 we ran three contests involving a total of 116 teams and two 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; build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also build-it teams were significantly better at finding security bugs. We plan to freely release BIBIFI to support future research. We believe it can act as an incubator for ideas to improve secure development. More information, data, and opportunities to participate are available at https://builditbreakit.org

8. REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC) 13*, 1 (2009), 4:1–4:40.

[2] The ACM-ICPC International Collegiate Programming Contest. http://icpc.baylor.edu.

[3] Burnham, K. P., Anderson, D. R., and Huyvaert, K. P. AIC model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons. *Behavioral Ecology and Sociobiology 65*, 1 (2011), 23–35.

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

[5] DEF CON Communications, Inc. Capture the flag archive. https://www.defcon.org/html/links/dc-ctf.html.

[6] dragostech.com inc. Cansecwest applied security conference. http://cansecwest.com.

[7] Edmundson, A., Holtkamp, B., Rivera, E., Finifter, M., Mettler, A., and Wagner, D. An empirical study on the effectiveness of security code review. In *International Symposium on Engineering Secure Software and Systems (ESSoS)* (2013).

[8] Finifter, M., and Wagner, D. Exploring the relationship betweenweb application development tools and security. In *USENIX Conference on Web Application Development (WebApps)* (2011).

[9] Git – distributed version control management system. http://git-scm.com.

[10] Google code jam. http://code.google.com/codejam.

[11] Harrison, K., and White, G. An empirical study on the effectiveness of common security measures. In *Hawaii International Conference on System Sciences (HICSS)* (2010).

[12] ICFP programming contest. http://icfpcontest.org.

[13] Inc, D. C. C. Def con hacking conference. http://www.defcon.org.

[14] Kim, Q. Want to learn cybersecurity? head to def con. http://www.marketplace.org/2014/08/25/tech/want-learn-cybersecurity-head-def-con, 2014.

[15] Maryland cyber challenge & competition. http://www.fbcinc.com/e/cybermdconference/competitorinfo.aspx.

[16] National Collegiate Cyber Defense Competition. http://www.nationalccdc.org.

[17] Parker, J. LMonad: Information flow control for haskell web applications. Master's thesis, Dept of Computer Science, the University of Maryland, 2014.

[18] Polytechnic Institute of New York University. Csaw - cybersecurity competition 2012. http://www.poly.edu/csaw2012/csaw-CTF.

[19] Prechelt, L. Plat_forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering 37*, 1 (2011), 95–108.

[20] PostgreSQL: The world's most advanced open source database. http://www.postgresql.org.

[21] Ruef, A., Hicks, M., Parker, J., Levin, D., Memon, A., Plane, J., and Mardziel, P. Build it break it: Measuring and comparing development security. In *CSET* (2015).

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

[23] Scandariato, R., Walden, J., and Joosen, W. Static analysis versus penetration testing: A controlled experiment. In *IEEE International Symposium on Reliability Engineering (ISSRE)* (2013).

[24] Stefan, D., Russo, A., Mitchell, J., and Mazieres, D. Flexible dynamic information flow control in haskell. In *ACM SIGPLAN Haskell Symposium* (2011).

[25] Top coder competitions. http://apps.topcoder.com/wiki/display/tc/Algorithm+Overview.

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

[27] Walden, J., Stuckman, J., and Scandariato, R. Predicting vulnerable components: Software metrics vs text mining. In *IEEE International Symposium on Software Reliability Engineering* (2014).

[28] Yang, J., Ryu, D., and Baik, J. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *International Conference on Big Data and Smart Computing (BigComp)* (2016).

[29] Yesod web framework for haskell. http://www.yesodweb.com.