Build It Break It: Measuring and Comparing Development Security

Andrew Ruef, Michael Hicks, James Parker, Dave Levin,
Atif Memon, Jandelyn Plane, and Piotr Mardziel
University of Maryland, College Park

Abstract

There is currently little evidence about what tools, methods, processes, and languages lead to secure software. We present the experimental design of the Build it Break it secure programming contest as an aim to provide such evidence. The contest also provides education value to participants where they gain experience developing programs in an adversarial settings. We show preliminary results from previous runs of the contest that demonstrate the contest works as designed, and provides the data desired. We are in the process of scaling the contest to collect larger data sets with the goal of making statistically significant correlations between various factors of development and software security.

1 Introduction

Experts have long advocated that achieving security in a computer system requires careful design and implementation from the ground up [19]. Over the last decade, Microsoft [9, 10] and others [23, 14, 12] have described processes and methods for building secure software, and researchers and companies have, among other activities, developed software analysis and testing tools [7, 5, 13] to help developers find vulnerabilities. Despite these efforts, the situation seems to have worsened, not improved, with vulnerable software proliferating despite billions spent annually on security [3].

This disconnect between secure development innovation and marketplace adoption may be due to a lack of *evidence*. In particular, no one would disagree that certain technologies, such as type-safe programming languages and static analysis tools, and activities, such as code reviews and penetration testing, can improve software's security. The question is how *effective* these (and other) things are, in terms of their costs vs. benefits. Companies would like to know: Which technologies or processes shake out the most (critical) bugs? Which are the easiest to use? Which have the least impact on other metrics of software quality, such as performance and maintainability? It is currently difficult to answer these questions because we lack the empirical evidence needed to do so.

In this paper we present **Build-it, Break-it, Fix-it** (BiBiFi), a new software security contest that is a fun experience for contestants, as well as an experimental testbed by which researchers can gather empirical evidence about secure development. 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, fast, and feature-ful. The second phase, *Break-it*, asks teams to find security violations and other problems in other teams' build-it submissions. Identified problems, proved via test cases, benefit a Break-it team's score while penalizing the Build-it team's score. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, gives builders a chance to fix bugs and thereby possibly get some points back if the process discovers that distinct break-it test cases identify the same underlying software problem.

We designed BiBiFi to give researchers *in situ* data on how teams develop code—for better or for worse—when security is a first-order design goal. As such, BiBiFi contestants are afforded a significant amount of freedom. Build-it teams can choose to build their software using whatever programming language(s), libraries, tools, or processes they want, and likewise Break-it teams can hunt for vulnerabilities as they like, using manual code reviews, analysis or testing tools, reverse engineering, and so on. During the contest, we collect data on each team's activities. For example, we keep track of commit logs, testing outcomes, and interactions with the contest site. Prior to and after the contest, we survey the participants to record their background, mindset, choices, and activities.

By correlating *how* teams worked with *how well* they performed in the competition, researchers can learn about what practices do and do not work. Treating the contest as a scientific experiment, the contest problem is the control, the final score is the dependent variable, and the various choices made by each team, which we observe, are the independent variables. To permit repeatable, statistically significant results, we have designed BiBiFi and implemented its infrastructure to scale to multiple competitions with thousands of competitors.

Our hope is to use BiBiFi—and for other researchers to apply it, as well—to empirically learn what factors most strongly influence security outcomes: Does better security correlate with: the choice of programming language? the use of development or analysis tools? the use of code reviews? participants' background (e.g., certification) and education (e.g., particular majors or courses)? the choice of a particular build process, or perhaps merely the size of the team itself? As educators, we are particularly interested in applying answers to questions like these to better prepare students to write responsible, secure code.

This paper makes two main contributions. First, it describes the design of BiBiFi (Section 2) and the contest infrastructure (Section 3), focusing on how both aim to ensure that high scores in the contest correlate with meaningful, real-world outcomes.

Second, it describes our first run of the contest, held during the Fall of 2014 (Section 4). We describe the contest problem, the rationale for its design, and details of how the contest proceeded and concluded. While participation in this contest was modest, and thus we cannot view the data we gathered as significant, the outcomes were interesting, and taught us important lessons that can be applied to future runs.

To the best of our knowledge, BiBiFi is the first public, large-scale contest whose focus is on *building secure systems*. Our contest takes inspiration from popular capture-the-flag (CTF) cybersecurity contests [15, 6, 17] and from programming contests [21, 2, 11], but is unique in that it combines ideas from both in a novel way. Although no contest can truly capture the many difficulties of professional development, we hope it will provide some general insight nevertheless, and thus provide some sorely needed evidence about what works for building secure software.

2 Build-it, Break-it, Fix-it: Design

This section summarizes the BiBiFi contest goals and how the contest's design satisfies those goals.

2.1 Goals, incentives, and disincentives

Our most basic goals 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. We break down these high-level goals into a set of 12 concrete requirements that guide our design.

The winning Build-it team would ideally develop software that is (1) **correct**, (2) **secure**, (3) **efficient**, and (4) **maintainable**. While the quality of primary interest to our contest is security, the other aspects of quality cannot be ignored. It is easy to produce secure software that is incorrect (have it do nothing!). Ignoring performance is also unreasonable, as performance concerns are often cited as reasons not to deploy defense mechanisms [1, 22]. Maintainability is also important—spaghetti code might be harder for attackers to reverse engineer, but it would also be harder for engineers to maintain. So we would like to encourage build-it teams to focus on all four aspects of quality.

The winning Break-it team would ideally (5) **find as many defects as possible** in the submitted software, thus giving greater confidence to our assessment that a particular contestant's software is more secure than another's. To maximize coverage, each break-it team should have incentive to (6) **look carefully at multiple submissions**, rather than target a few submissions. Moreover, teams should be given incentive to, in some cases, (7) **explore code in depth**, to find tricky defects and not just obvious ones. We would like break-it teams to (8) **produce useful bug reports** that can be understood and acted on to produce a fix. Finally, we would like to (9) **discourage superficially different bug reports** that use different inputs to identify the same underlying flaw.

We also have several general goals. First, we want to (10) scale up the contest as much as possible, allowing for up to several hundred participating teams. The more teams we have, the greater our impact from both a teaching and research point of view: on the teaching side, more students experience the desired learning outcomes, and on the research side, we have a greater corpus of data to draw on. To support scalability, the contest design should (10a) **encourage greater participation**, and (10b) **minimize the amount of manual judgment** required to assess the results. The latter is important for reducing staffing needs and also helps make the outcomes more understandable. We also wish to (11) **discourage collusion** where it might provide an unfair advantage. As a final goal, we want (12) the data that we gather from running the contest—the activity logs and submissions produced by the participants, as well their metadata—to be useful in assessing the reasons for a team's success or failure.

2.2 Contest structure

BiBiFi begins with the **build-it phase**. Registered contestants aim to implement the target software system according to a published specification. A suitable target is one that can be completed by good programmers in a short time (from a weekend up to a couple of weeks, depending on phase length), has an interesting attack surface, and is easily benchmarked for performance; we discuss these points in greater depth in Section 2.4. The implementation must build and run on a standard Linux VM made available prior to the start of the contest. Teams may be geographically distributed—the competition is not hosted on-site—and develop using Git [8]; with each push, the contest infrastructure downloads the submission, builds it, and tests it (for correctness and performance), updating the scoreboard.

The next phase is the **break-it phase**. Break-it teams can download, build, and inspect all build-it submissions, including source code.[1] We randomize each break-it team's view of the build-it teams' submissions, but organize them by metadata like programming language. When they think they have found a defect, they submit a test case. BiBiFi's infrastructure accepts this as a defect if it passes our reference implementation but fails on the buggy implementation. Any failing test is potentially worth points; we take the view that a correctness bug is potentially a security bug waiting to be exploited. That said, additional points are granted to an actual exploit or to demonstrated violations of security goals; for the latter we require special test case formats (see Section 4). To encourage coverage, a break-it team may submit up to ten passing test cases per build-it submission.

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 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.

Inspired by the ICFP programming contest [11], we had originally set the contest to run on three consecutive (3-day) weekends, but found that this was too aggressive (too few teams qualified). We plan to set them to 1–2 weeks each, in our next run.

Scoring in BiBiFi works as follows. As the contest unfolds we post each team's build-it and break-it scores. Each build-it team receives a fixed point total for passing a core suite of correctness tests, relative to the number of qualifying submissions M; e.g., $5M$ points. Teams get a linearly scaled bonus according to their rank-ordering, per performance test; i.e., the best performing submission for a given test earns M points, while the worst earns 0. Teams receive extra points for successfully implementing optional features from the problem specification; each optional test is worth $M/2$ points. After the break-it and fix-it phases, we will know the set of unique defects against a submission, and which teams reported those defects. A unique defect deducts a fixed point value P from a build-it team's initial score, and each of the N break-it teams that identified the defect will gain P/N points to their break-it score. For correctness bugs, we set P

[1] We check that submissions contain source code that builds properly and has not been mechanically obfuscated [4].

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 $2M$. After the fix-it phase, and all scores are final, the top scorers for both the build-it and break-it categories are awarded prizes.

2.3 Assessment

Consider this design with respect to the goals outlined in Section 2.1, starting with the build-it teams. Because they will lose points for each discovered defect, they have incentive to write correct code (1)(2). Moreover, granting more points to the fastest implementations provides incentive to write efficient implementations (3). Finally, it is in the participants' best interests to write maintainable (i.e., understandable) code because they can gain points back by fixing flaws provided in the final phase (4). It is important for the build-it phase to not be too long, thus forcing teams to carefully weigh their options—to maximize points they need to pick a method that will allow them to build the system quickly, correctly, and securely.

The break-it teams are also encouraged 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 (8). Because they are limited to a fixed number of test cases per submission, they have incentive to examine as many submissions as possible, to gain more points (6). Moreover, they are incentivized to ensure that test cases for any given submission are not morally the same, or else they will not gain as many points (9). Finally, by providing fewer points for defects also found by other teams, break-it teams are encouraged to look deeper for hard-to-find bugs, rather than just low-hanging fruit (7). Together, these incentives encourage both broad and deep exploration to find many unique bugs (5).

Now consider the final set of goals. To encourage participation (10a), contestants may participate remotely, commit only a short amount of time, receive online accreditation, and win cash prizes (and notoriety). To reduce the amount of work for the organizers (10b), we employ best-effort automation and incentivize teams to make the organizers' jobs easy. In particular, rather than ask organizers (or automation) to perfectly disambiguate similar defects, build-it teams will do it (and break-it teams will avoid testing their ability to do it) because it is in their interests. The organizers simply confirm that the teams are not fixing multiple bugs at once, which teams are motivated to make clear so their fixes are approved.

Discouraging collusion (11) is crucial to ensuring fair outcomes. There are three broad possibilities for collusion, each of which we discourage with how we score and administer the competition.

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.

2.4 Discussion: Task Selection

For each offering of the contest, we must design a suitable programming task, implement a reference solution, and write basic performance and correctness tests. To ensure that the contest outcomes are meaningful, we believe the task must exhibit the following five qualities:

First, it must be *appropriately sized.* The software system must be constructible within a short time window by a small, motivated team. Second, the implemented software must have *measurable per-*

formance to motivate programmers to take risks in their design to make it efficient. Third, the task should require *independent development.* The task should not be too similar to an existing software to ensure that contestants do the coding themselves. (Of course, we expect contestants to use libraries and frameworks.)

These three qualities are typical for many coding competitions (and class projects). What distinguishes BiBiFi tasks are their security-related goals:

Our fourth goal is that a task should have a *large attack surface*, meaning that there is a sufficiently rich set of security-relevant interactions to potentially exploit. Finally, the task's functionality should be *relevant:* it must represent, in terms of application area and underlying code structure, a class of systems that are security-critical, so that the outcomes are meaningful to the question of building real-world secure systems.

We describe in Section 4.1 the task we used in our first BiBiFi run (a secure, append-only log).

3 Contest implementation

Figure 1 illustrates components of the contest implementation (along the left), and the interactions between participants and outside elements (on the right). This section describes key implementation details that aim to ensure security and scalability.

3.1 BIBIFI web application

The locus of the BIBIFI contest is the web application running at `https://builditbreakit.org`. Contestants sign up for the contest using this application, filling out a survey when doing so, to gather demographic data and other data potentially relevant to the contest outcome (e.g., programming experience and security training). During the contest, the web app orchestrates the key functions: It tests build-it submissions and break-it bug reports; it keeps the current scores updated; and it provides a workbench for the judges for considering whether or not a submitted fix covers one bug or not.

It is important that the web application be secure; otherwise, contest outcomes could be thwarted by unscrupulous participants. To support this end, we implemented the web application in about 9000 lines of Haskell using the Yesod [25] web framework backed by a PostgreSQL [18] 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 [16], which is derived from LIO [20], in order to protect against inadvertent information leaks and privilege escalations. LMonad dynamically guarantees that users can only access their own information.

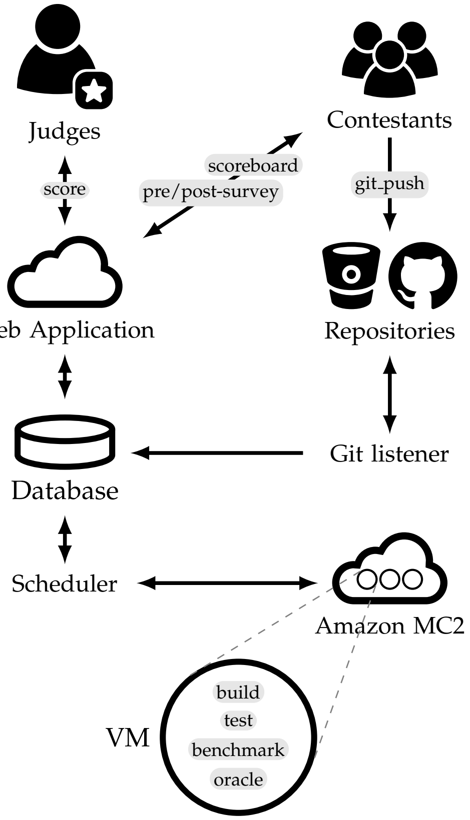

Figure 1: Implementation overview.

3.2 Testing submissions

Our implementation includes backend infrastructure for testing, both during the build-it round for correctness and performance, and during the break-it round to assess potential vulnerabilities. The backend consists of approximately 4400 lines of Haskell code (and a little Python).

To implement testing we require contestants to specify a URL to a Git [8] 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. Each time this happens, the backend 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. The web application randomizes the order the submissions are presented to each break-it team to encourage broad coverage. 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 this file (to their Git repository) and then push. The backend uses the file to set up a test of the implicated submission to see if it indeed is a bug. How it does this depends on the specification and the kind of bug.

4 Experience

We have run BiBiFi twice: once as a pilot open only to contestants at our institution(s), in early 2014, and later for all US college students, in Fall 2014. Participant data was anonymized and stored in a manner approved by the UMD IRB. Participants consented to have data related to their activities collected, anonymized, stored, and analyzed. This section describes the Fall 2014 contest, some of the data we collected, and some lessons learned.

4.1 Contest problem

The programming problem for the contest was to implement a *secure log* for a hypothetical art gallery alarm system. This log stores events generated by sensors when employees and visitors enter and exit gallery rooms. Two programs, logappend and logread, are used to access the log. The former adds new events to the log, while the latter reads the log to query about certain events. An empty log is created by logappend with a given authentication token, and all subsequent calls to logappend and logread on the same log must use that token.

Build-it teams could design the log format and implement these two programs however they like while ensuring both log *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 detected. Performance was measured in terms of time to perform a particular sequence of operations, and space consumed by the resulting log.

Break-it teams could demonstrate correctness bugs by showing that a program did not adhere to the specification. They could also demonstrate violations of confidentiality or integrity as follows. When providing submissions to the break-it teams, we also included a set of log files. In some cases we just provided the file, and in other cases we also provided a transcript of the logappend operations used to generate the file, but omitting the token used. A break-it team could submit a test case involving a transcript-less log that demonstrates knowledge of the file's contents, thus violating privacy. A team could also submit a test case involving the second kind of log, where using logread on a modified version of the log (also provided by the team) outputs a different answer than with the unmodified log (when using the correct token), thus violating integrity.

4.2 Outcomes and Lessons

A total of 90 teams registered for the contest. Of these, 28 teams (totaling 64 people) shared a Git repository with us and carried out some development. Of these, 11 qualified to be break-it round targets.[2]

Nine teams (25 people) competed in the break-it round. Of these, seven teams scored points by finding bugs in build-it submissions. We also had two professional teams (from Cyberpoint and AT&T) participate in the Break-it round. The bugs they found counted against the Build-it scores, but did not affect student Break-it teams' scores. In total, break-it teams found 178 unique bugs. The

[2] A twelfth team (121) had a working submission until five minutes before the submission deadline, but then pushed a version that "added compression" that failed to pass the correctness tests.

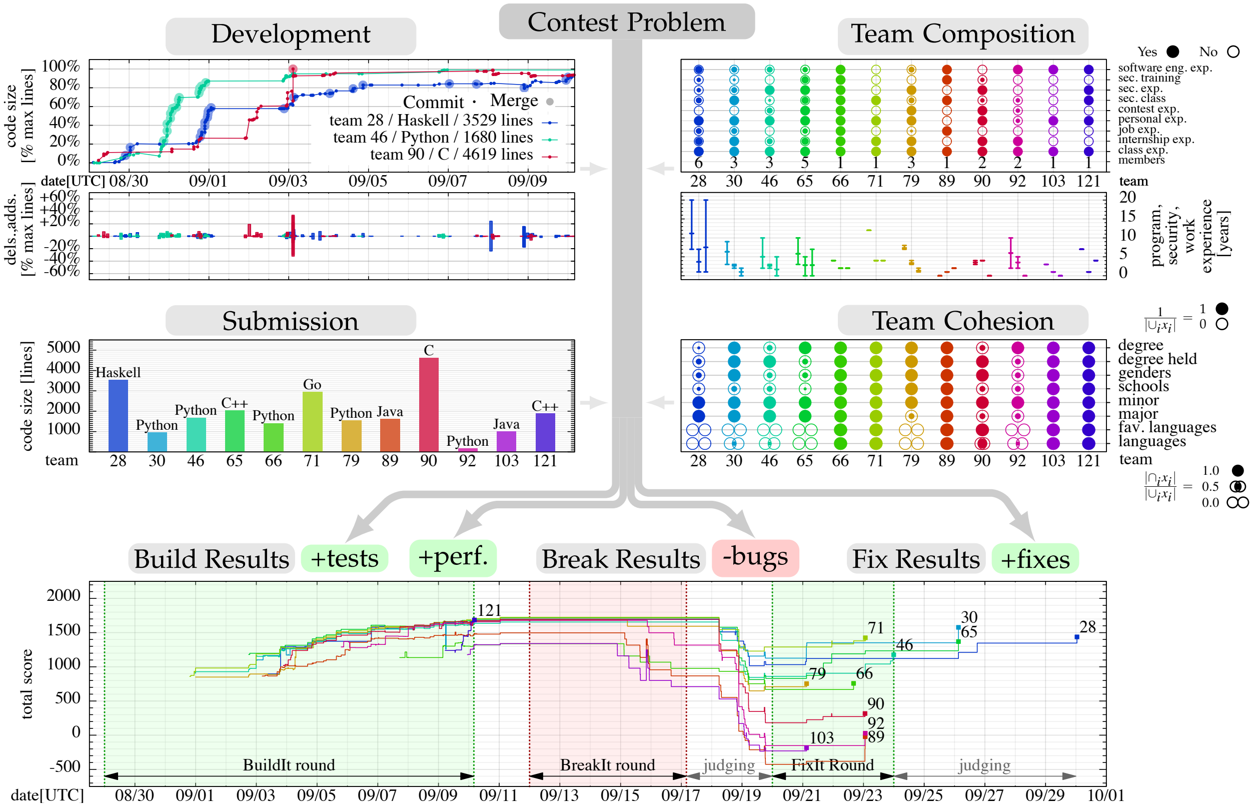

Figure 2: Fall 2014 contest data sample. A sample of three teams' development data (top left) includes the git commit history featuring code changes and merges (large shaded dots). The magnitude of the code changes is visualized both as code lines added and code lines deleted as a proportion of maximum lines The final qualifying submissions (middle left) are shown along with their language and size. The team composition (top right) derived from pre/post contest surveys includes 1) yes/no questions and 2) years of experience. Additionally the surveys provide statistics relating to the cohesion of the teams (middle right). This includes single answer questions which are visualized as the inverse of the set size of the team's answers (top 6 lines), and multiple answer questions which are visualized as the size of the intersection as a proportion of the union (bottom 2 lines).

winning student teams did better than both professional teams. In general, student teams largely employed testing and manual inspection to find bugs. The top break-it team was also the third-place build-it team, and reused its build-it test suite to find bugs in others' submissions. The second-place team and professional teams also used static analysis and fuzzing.

Three build-it teams earned points back during the fix-it round. Doing so dropped the total number of unique bugs from 200 to 178.

The bottom of Figure 2 shows the contest as it unfolded, plotting the score of each qualifying build-it teams across all three rounds (build-it, break-it, and fix-it). In terms of final scores, the top three build-it teams were teams 30 (1st), 28 (2nd), and 71 (3rd), while the top break-it teams were teams 71 (1st), 66 (2nd), and 114 (3rd). Other data we gathered during this contest run is visualized in the top half of the figure and described in the caption.

Lessons learned. The Fall 2014 results confirmed the importance of having a contest problem with specific security goals. Our pilot contest asked contestants to build a secure parser (for SDXF [24]); as parsers are often a target of exploits, we felt this topic was relevant. However, writing a secure parser requires no real ingenuity: write it in a type- or memory-safe language. This is what the builders did, leaving nothing on the table for break-it teams. Our Fall 2014 problem had specific privacy and integrity goals that we could directly test for, and for which type- and memory-safety is not sufficient. Instead, contestants needed to include some sort of cryptography in their design. Some teams failed to do so, relying on the obscurity of their runtime's object serialization protocol to deter attacks. Others had errors with the use of a cryptographic library, or with not validating the integrity of the log file as a whole. Break-it teams found and exploited these flaws, resulting in a large drop in score for several of the submissions. Only one submission written in C had a heap overflow error in command line parsing and the application could be made to segfault from command line arguments.

We also learned that timing and phase duration are extremely important: We had originally set the build-it round at three days, but only one team qualified! Therefore we extended the round to nearly two weeks (explaining the change in commit activity shown the upper left of Figure 2).

Another lesson was that correctness bugs can have an outsized effect without a comprehensive test suite; correctness bugs had a big impact on final scores. Moreover, correctness bugs need an unambiguous judge—an oracle implementation—since the specification is inevitably going to be vague; several "fixes" at the end were allowances due to specification ambiguity.

5 Conclusions

We have designed, developed, and held initial runs of a contest, BiBiFi, that is a fun arena for contestants, and an experiment testbed for researchers. BiBiFi's design seeks to align the incentives of the contestants (who wish to win) with those of researchers (who wish to see what leads to the best code and the most thorough attack analysis). Our hope is that BiBiFi can facilitate good educational experiences and supply empirical evidence about best practices, both for building secure code and breaking insecure code. More information, data, and opportunities to participate are available at `https://builditbreakit.org`

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 4:1–4:40.

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

[3] Anderson, R., Barton, C., Bóhme, R., Clayton, R., van Eeten, M., Levi, M., Moore, T., and Savage, S. Measuring the cost of cybercrime. In *Proceedings of the 11th Annual Workshop on the Economics of Information Security* (2012).

[4] Collberg, C., and Thomborson, C. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *Software Engineering, IEEE Transactions on 28*, 8 (aug 2002), 735 – 746.

[5] Coverity security advisor. `http://www.coverity.com/products/security-advisor.html`.

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

[7] Fortify software security center. `http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center`.

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

[9] Howard, M., and LeBlanc, D. *Writing Secure Code*. Microsoft Press, 2003.

[10] Howard, M., and Lipner, S. *The Security Development Lifecycle*. Microsoft Press, 2006.

[11] ICFP programming contest. `http://icfpcontest.org/`.

[12] ISC(2). (CSSLP) certification. `http://www.isc2.org/csslp`.

[13] Owasp lapse project. `https://www.owasp.org/index.php/Category:OWASP_LAPSE_Project`.

[14] McGraw, G. *Software Security: Building Security In*. Software Security Series. Addison-Wesley, 2006.

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

[16] Parker, J. L. LMonad: Information flow control for haskell web applications. Master's thesis, University of Maryland, College Park, 2014.

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

[18] PostgreSQL: The world's most advanced open source database. `http://www.postgresql.org/`. Accessed: 2015-05-01.

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

[20] Stefan, D., Russo, A., Mitchell, J. C., and Mazières, D. Flexible dynamic information flow control in haskell. *SIGPLAN Not. 46*, 12 (Sept. 2011), 95–106.

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

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

[23] Viega, J., and McGraw, G. *Building Secure Software: How to Avoid Security Problems the Right Way*. Professional Computing Series. Addison-Wesley, 2001.

[24] Wildgrube, M. Structured data exchange format (SDXF). Tech. Rep. RFC 3072, Network Working Group, May 2001.

[25] Yesod web framework for haskell. `http://www.yesodweb.com/`. Accessed: 2015-05-01.