Build It, Break It, Fix It: Contesting Secure

Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. M
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 BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teamsâĂŹ submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. During 2015 we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in 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.

1. INTRODUCTION

Capture-the-flag (CTF) and other cybersecurity contests 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 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 , not something that can be added after the

other teams' build-it submis
via test cases, benefit a br
ize the build-it team's scor
security-relevant problems.
it scores are independent,
each category.) The final
fix bugs and thereby get po
ers that distinct break-it t
fect. BIBIFI's structure ar
designed to encourage mean
that the top-scoring build-it
efficient software. Behavior
comes are discouraged. Fo
submit a limited number o
mission, and will lose points
expose the same underlying
by other teams. As such, t
bugs broadly (in many subm
hard-to-find bugs).

In addition to providing
BIBIFI presents an opportu
breaking process scientifica
tests may serve as a quasi-
relates participation data w
ing artifacts and participant
choice of build-it programm
perience, code size, testing
team's (non)success in the
To the extent that contest
test participants represent
munity, the results of this stu
evidence for practices that h
Indeed, the contest environ
ideas to improve developme
making their way to practic

This paper studies the ou
that we held during 2015, i
ming problems. The first co
a *secure, append-only log* f

Build It, Break It, Fix It: Contesting Secure

Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. M
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 BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teamsâĂŹ submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. During 2015 we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in 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.

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

other teams' build-it submis
via test cases, benefit a br
ize the build-it team's scor
security-relevant problems.
it scores are independent,
each category.) The final
fix bugs and thereby get po
ers that distinct break-it t
fect. BIBIFI's structure ar
designed to encourage mean
that the top-scoring build-it
efficient software. Behavior
comes are discouraged. Fo
submit a limited number o
mission, and will lose points
expose the same underlying
by other teams. As such, t
bugs broadly (in many subm
hard-to-find bugs).

In addition to providing
BIBIFI presents an opportu
breaking process scientifica
tests may serve as a quasi-
relates participation data w
ing artifacts and participant
choice of build-it programm
perience, code size, testing
team's (non)success in the
To the extent that contest
test participants represent
munity, the results of this stu
evidence for practices that h
Indeed, the contest environ
ideas to improve developme
making their way to practic

This paper studies the ou
that we held during 2015, i
ming problems. The first co
a *secure, append-only log* f
generated by a hypothetical
tackers with direct access to

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 anal-

2.1 Competition pha

We begin by describing t
occurs during a BIBIFI co
ministered on-line, rather th
graphically distributed. The
each of which last about tw
scribe in this paper.

BIBIFI begins with the **b**
testants aim to implement
cording to a published spec
administrators. A suitable
pleted by good programme
two weeks, for the contests
for performance, and has an
software should have specif
ing private information or c
compromised by poor desig
software should also not be
to ensure that contestants
still taking advantage of hi
works to the extent possible
run on a standard Linux V
start of the contest. Teams
each push, the contest infras
sion, builds it, tests it (for
and updates the scoreboar
problems we developed: (1)
pair of communicating prog
ATM.

The next phase is the **br**
can download, build, and in
missions, including source
must build properly, pass a
purposely obfuscated. We
view of the build-it teams' s
by meta-data like programm
they have found a defect, bre
poses the defect and an expl
age coverage, a break-it tea
number of test cases per bu
frastructure automatically j
case truly reveals a defect.
bug, it will run the test aga
("the oracle") and the targe
test passes on the former b
accepted.[3] More points are
lems, which may be demons
mats. The auto-judgment a
two different contest proble

The final phase is the **fix**
provided with the bug rep
their submission. They may
tify; if a single fix corrects
the test cases are "morally
only deducted for one of th

ysis 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).

[1]This paper subsumes a previously published 6-page workshop paper [redacted]. 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.

based on information provi
other assessment, whether a
sense that it corrects only

[2]This avoids spurious unfair
investigating code in the orde
[3]To encourage testing of our
earn points by finding bugs in

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

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 [redacted], 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

spirit of the competition. I
vide test cases as evidence
ensure they are providing u
4× more points for security-
teams to look for these sort
on correctness issues. (But
ter; see below.) Because
fixed number of test cases p
aged from submitting many
the same;" as they could
fix-it phase they are better
ing different bugs. Limiting
encourages examining many
points for defects found by
teams are encouraged to lo
than just low-hanging fruit.

2.2.3 Limitations

While we believe BIBIFI
tives are properly designed,
itations.

First, there is no guaran
fects will be found. Break-
skill to find problems in all
sions may receive equal scr
act contrary to incentives a
duplicated bugs, rather than
Finally, break-it teams may
frastructure cannot automa
defects will go unreported.
pool of break-it teams, and
dations automation, we stil
in breadth and depth.

Second, builders may fail
in their best interests. For
a fix rejected as addressing
teams may use several speci
would have been allowed.
of contention for prizes ma
fix-it phase.[5] We observed
described in §4.5. Both acti
score (and correspondingly
can mitigate these issues wi
e.g., by offering prizes to all
their final score, rather than

Finally, there are several
inition that may skew result
ness tests may leave too ma
during break-it (distracting
security issues); too many
few (meaning teams are dif
eral bug-finding ability). S
lems 4 to 1 over correctness
what ratio makes the most
real-world outcomes; both

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

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

argued. Finally, performan
portant design tradeoffs (e.g
ways that teams approach m
the contests we report in th
able with these design point

[5]Hiding scores during the con
would harm incentives during
with no bugs reported against

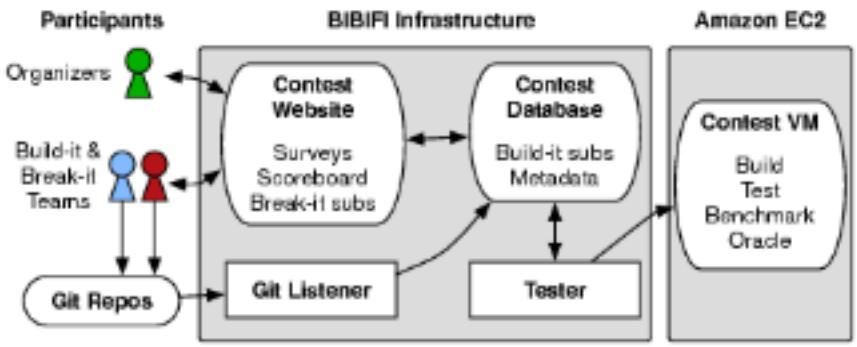


Figure 1: Overview of BIBIFI's implementation.

test [redacted] 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 participation, but it also opens the possibility of collusion between teams, as there cannot be a judge overseeing their communication and coordination. There are three broad possibilities for collusion, each of which BIBIFI's scoring discourages.

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

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

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

2.3 Implementation

Figure 1 provides an overview of the BIBIFI implementation. It consists of a web frontend, providing the interface

evant to the contest outcom
and security training). Dur
cation tests build-it submis
keeps the current scores upd
for the judges for considerin
covers one bug or not.

To secure the web applic
ticipants, we implemented it
the Yesod [29] web framewo
database. Haskell's strong t
after-free, buffer overrun, a
attacks. The use of Yesod
tion against various attacks
jection. As one further lay
tion incorporates the infor
LMonad [redacted], which is deri
protect against inadvertent
escalations. LMonad dynam
only access their own inform

Testing backend. The bac
testing during the build-it
formance, and during the b
vulnerabilities. It consists
(and a little Python).

To automate testing, we
a URL to a Git [redacted] reposit
Bitbucket, and shared with
read-only. The backend "lis
tory for pushes, upon which
commit. Testing is then ha
up an Amazon EC2 virtual
each submission. We requi
runs, without any network
that we share in advance.
stall additional packages no
VMs supports both scalabil
dynamically provisioned) a
stances prevents a team fro
tests, or of tests on other te

All qualifying build-it su
by break-it teams at the s
break-it teams identify bug
file specifying the buggy su
of commands with expected
bug. Break-it teams commi
repository). The backend us
implicated submission to se

3. CONTEST PROB

This section presents the
developed for the contests he
specific notions of security d
such defects were tested.

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 rel-

3.1 Secure log (Sprin

The secure log problem w
art gallery security system.
The first, `logappend`, appen
indicate when employees an
rooms. The second, `logre`
events. To qualify, submiss
queries (involving the curre

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 authenti-

vacy break against the sco
even if there are multiple
to produce privacy/integrity
automatically tell them apa

3.2 Securing ATM in

The ATM problem asked
municating programs: `atm`
customers to set up an acco
money, while `bank` is a serve
tracking bank balances. `at`
a customer with a correct
balance of their account, a
(e.g., they may not withdra
In addition, `atm` and `bank` s
can authenticate each other
this purpose; it will be share
channel unavailable to the a
municating with `bank` over t
dle" (MITM) could observe a
or insert new messages. The
security despite not having

A canonical way of imple
grams would be to use pub
encrypted communications.
`bank`'s public key to ensure t
the `atm` is with the `bank` and
an account, the card file sho
number, so that the MITM
It is also necessary to prote
ing nonces or similar mecha
wise approach would be use
ment these features. Both
are discussed further in §5.

Build-it submissions' per
time to complete a series o
`atm`/`bank` interactions.[7] Cor
were defined as sequences o
geted submission produces
(or crashes). Security defect

Integrity breaks. Integrity
ing a custom MITM progra
on a specified IP address
connection from the `atm` wh
MITM program can thus
nications between `atm` and
or initiate its own. We pro
a starter MITM: It sets up
communications between th

To demonstrate an integri
requests to a *command serv*
inputs on the `atm` and it c
account whose creation it i
will declare the test complet

cation 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 pri-

`atm` commands is run using t
the MITM. This means tha
sends directly to the target s

[6]In a real deployment, this mi
file into the ATM's ROM prio
[7]This transcript was always s
vation to support parallelism f
[8]All submissions were require

be replayed/sent to the oracle. If the oracle and target both complete the command list without error, but they differ on the outputs of one or more commands, or on the balances of accounts at the bank whose card files were not revealed to the MITM during the test, then there is evidence of an integrity violation.

As an example (based on a real attack we observed), consider a submission that uses deterministic encryption without nonces in messages. The MITM could direct the command server to 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 implementa-

performance in each phase
factors contribute to high s
ing by other teams, and st
We find that on average, te
other than C and C++, a
more programming languag
programming skill), are less
tified in their code. Success
identifying security bugs in
with having more team mem
ing in the build-it phase (an
to how to secure an impleme
contest, which used the ATI
more security bugs than the

4.1 Data collection

For each team, we collecte
reported data. When signi
ported standard demograph
experience and programmin
contest, each team member
about their performance. I
mation about lines of code w
from teams' Git repositories

Participant data was ano
approved by our institution'
Participants consented to h
ties collected, anonymized, s
ticipants did not consent to
personal data was not used

4.2 Analysis approac

To examine factors that c
ing and breaking, we apply
sion model attempts to expl
one or more measured facto
participants' scores, we can
which estimates how many p
to (or takes away from) a t
outcomes, such as whether
we apply logistic regression.
each factor impacts the like

We consider many variabl
teams' results. To avoid ov
potential factors those varia
interest, within acceptable
(Our choices are detailed be
els with all possible combina
select the model with the mi
terion (AIC) [redacted]. Only the f

We describe the results
not a completely controlled
random assignment), so our
rather than causation. Ou
confounds, and many factor

tions 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'

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

in our data are correlated
also assumes that the factor
on participants' scores (or or
while this may not be the
simplification for considerin

4.3 Contestants

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.

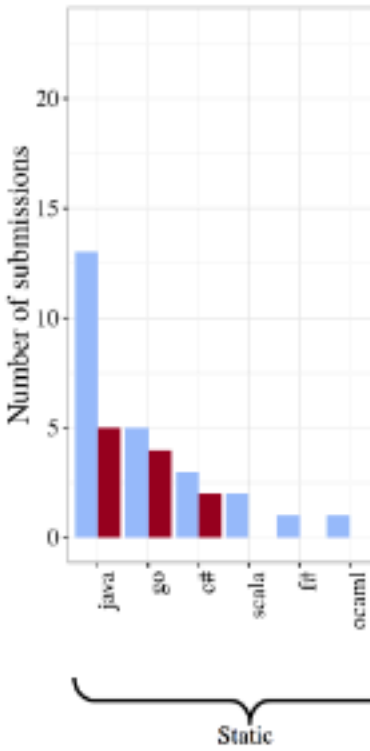


Figure 2: The number of b
test, organized by primary
The brackets group the lang

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 hardware 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

Model setup. To ensure e
ful relationships, we decide
size roughly equivalent to C
$f^2 = 0.15$. An effect t
explain up to 13% of the va
With an assumed power of
we limited ourselves to nine
a prospective $f^2 = 0.154$. (
model is reported with the r
this limit, we selected the fo

Contest: Whether the
Spring 2015 contest or the

 # Team members: A t

Knowledge of C: The
listed C or C++ as a progra
included this variable as a p
implementation details, a sk
for successful secure buildin

 # Languages known:
languages team members co
last row of Table 2). For e
where member A claims to
member B claims to know
the language count would b

Coding experience: Th
experience reported by a te

Lines of code: The SL
the team's final submission

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.

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

Language category: W
submission as having one "
guages were then assigned to
statically-typed (e.g., Java,
namically-typed (e.g., Perl,
category for the regression.
total submissions for each l

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

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.

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. 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)

Factor	
Fall 2015	
Lines of code	
Dynamically typed	-
Statically typed	
MOOC	

Table 3: Final linear regress
indicating how many points
total score. Overall effect si

the largest minimum size).

4.5 Code quality me

Now we turn to measures
in terms of its correctness an
up under scrutiny by break-

Resilience. The total build
just discussed, and *resilien*
score that derives from brea
the presence of defects. B
during the fix-it phase, as fi
cases that identify the same

Unfortunately, upon stud
large percentage of build-it
ported against their code, fo
doing so. We can see this
resilience scores (Y-axis) of
the two contests. The circle
fixed at least one bug, wher
that fixed no bugs. We ca
teams with the lower resilie
We further confirmed that
factor by running a regress
cluded fix-it phase participa

This situation is disappo
silience score as a good meas
to ship score). Our hypothe
sufficiently incentivized to f
teams that are sufficiently f
sen to fix no bugs because
for MOOC students, once a
were assured to pass; it m
necessary for attaining this
ing alternative structures th
teams to fix all (duplicated)

Presence of security bug
sufficiently meaningful, a us
that a build-it submission c
by this we mean any submi

in ship score. Based on our qualitative observations (see §6), 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

crash, privacy, or integrity
model we used logistic regres
as the ship model.

Table 4 lists the results o
efficients represent the cha
with each factor. Negative
hood of finding a security b
exponential of the coefficien
how strongly that factor b

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.

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.

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. Of the 16 C/C++ submissions (see Figure 2), 12 of them had a security bug: 5/9 for Spring 2015 and 7/7 for Fall 2015. All 5 of the buggy implementations from Spring 2015 had a crash

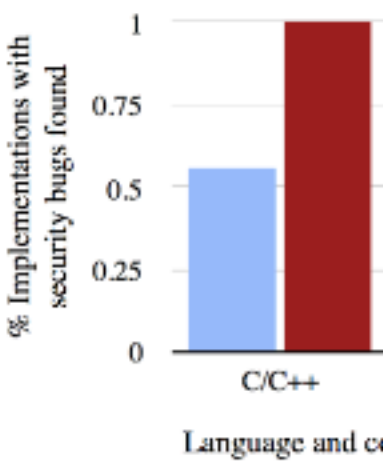


Figure 5: The fraction of tea
rity bug was found, for each

relevant and rerun the mode
language category is no long
may indicate that lack of m
vantage to using C/C++ fr
thus a memory-safe C/C++

Our model shows that tea
guages (even if they did not
mission) performed slightly
guage known. Additional L
also associated with a very s
an identified security bug.

Finally, the model shows
in the outcome, but not in
using a dynamically typed
the MOOC. We see the effec
the latter, the effect size is c
MOOC security training pla

4.6 Breaking success

Now we turn our attentio
i.e., how effective teams we
submissions. First, we con
formed as indicated by th
prior to the fix-it phase. W
raw output, ignoring wheth
bug (which we cannot assess
of fix-it phase participation
108 teams that participated
and Fall 2015. We also m
to **security bug count**, or
break-it team found. Doing
effort at finding correctness

We model both break-it s
ing several of the same pot
viously, but applied to the
building team. In particular

of the Buggy implementations from Spring 2015 had a crash defect, and this was the only security-related problem for three of them; none of the Fall 2015 implementations had crash defects. If we reclassify crash defects as non-security

[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.

participated in, whether the
number of break-it **Team m**
Coding experience, aver
of C, and unique **Languag**
members. We also add two

Build participant: Tru
pated in the build-it phase,

Advanced techniques:

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 $f^2 = 0.039$.

ported using software analysis or fuzzing to aid in bug finding. Teams that only used manual inspection and testing are categorized as false. 26 break-it teams used advanced techniques.

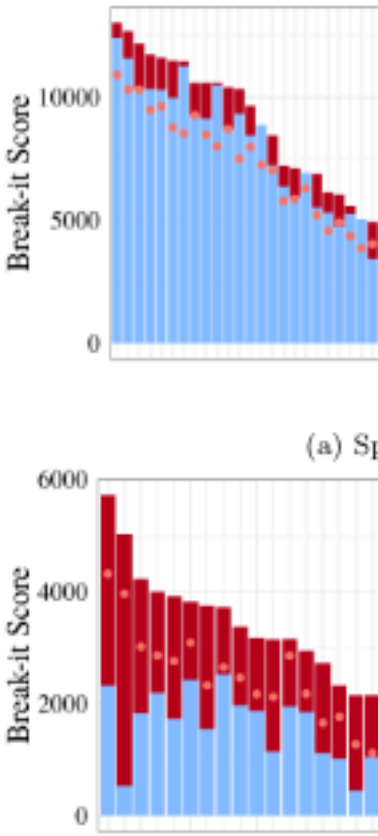
For these two initial models, our potential factors provide eight degrees of freedom; again assuming power of 0.75, this yields a prospective effect size $f^2 = 0.136$, indicating we could again expect to find effects of roughly medium size by Cohen's heuristic [redacted].

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 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 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*



(a) Sp

(b)

Figure 6: Scores of break-it broken down by points from The final score of the brea noted as a dot. Note the d general, the Spring 2015 co higher scores for breaking.

Factor
Fall 2015
Team members
Build participant

Table 6: Final linear regres curity bugs found by each t many security bugs each fa effect size $f^2 = 0.035$.

for each additional team me nificantly helps one's score one would expect to gain a system could fail after build shows the distribution of the per contest, for break-it tean it teams.

On average, four more sec

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

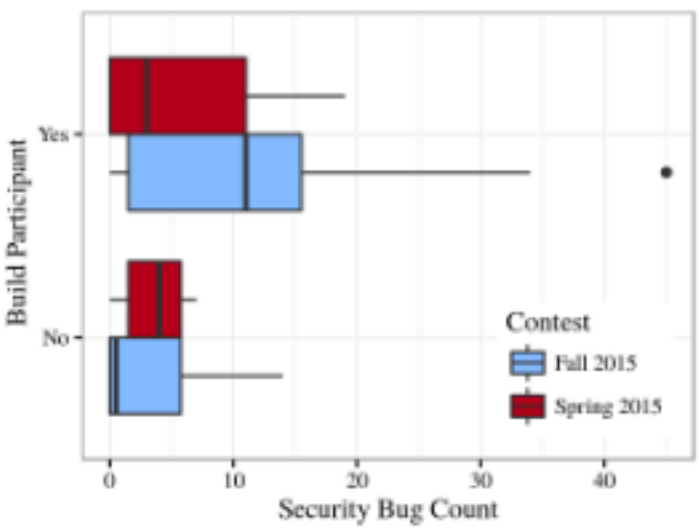


Figure 7: Count of security bugs found by each break-it team, organized by contest and whether the team also participated in build-it. The heavy vertical line in the box is the median, the boxes show the first and third quartiles, and the whiskers extend to the most outlying data within $\pm 1.5\times$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

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.

curity at all" to "vulnerable
tacks." This is interesting
observed in the software ma

Many implementations of
tion or authentication. Ex
trivial for break-it teams. S
as plain text, other times l
Java object serialization pro

One break-it team discov
could exploit with at most
mission truncated the "auth
vulnerable to a brute force

The ATM problem allows
sible for the log), and the
plementations used cryptog
One implementation used
RC4 from scratch and did
key or the cipher stream.
ciphertext of messages was
changed from transaction t
flip bits in a message to cha

Another implementation
cation, but did not use rand
were always distinguishable
was constructed against thi
tack leaked the bank balan
drawal attempts, distinguis
transactions, and performin

Some failures were comm
mentations: many implemen
the lifetime of the `bank` and
a nonce in the messages. T
messages freely between the
tegrity via unauthorized wit

Some failures were commo
well: if an implementation u
authentication. If it used au
cate records stored in the fi
The implementations would
tries in the file to the orde
for an integrity attack that
entries in the file.

6. RELATED WOR

BIBIFI bears similarity t
curity contests but is uniqu
systems. BIBIFI also is rela
development, but differs in

Contests. Cybersecurity co
ability discovery and exploi
system administration com
lar style of contest is dubb
is exemplified by a contest

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 se-

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 [redacted] asks teams to write several programs in C/C++ or Java during a 5-hour time period. Google Code Jam [redacted] sets tasks that must be solved in minutes, which are then graded according to development speed (and implicitly, correctness). Topcoder [redacted] 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 [redacted] and Prechelt [redacted] 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. [redacted] considered manual code review; Scandariato et al. [redacted] compared different vulnerability detection tools; other studies looked at software properties that might co-occur with security problems [redacted]. 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

act as an incubator for ideas
More information, data, and
available at https://builditb

8. REFERENCES

[1] Abadi, M., Budiu, M., E
Control-flow integrity pri
applications. *ACM Trans
System Security (TISSE*

[2] The ACM-ICPC Internat
Contest. http://icpc.bayl

[3] Burnham, K. P., Ander
AIC model selection and
ecology: some backgroun
Behavioral Ecology and S

[4] Cohen, J. *Statistical Pow
Sciences*. Lawrence Erlba

[5] DEF CON Communicat
archive. https://www.def

[6] dragostech.com inc. Ca
conference. http://cansec

[7] Edmundson, A., Holtka
M., Mettler, A., and W
the effectiveness of securi
*Symposium on Engineeri
(ESSoS)* (2013).

[8] Finifter, M., and Wagn
betweenweb application d
*USENIX Conference on
(WebApps)* (2011).

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

[10] Google code jam. http://

[11] Harrison, K., and Whit
effectiveness of common s
International Conference
(2010).

[12] ICFP programming conte

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

[14] Kim, Q. Want to learn cy
http://www.marketplace.
want-learn-cybersecurity-

[15] Maryland cyber challenge
fbcinc.com/e/cybermdcon

[16] National Collegiate C
http://www.nationalccdc

[17] Parker, J. LMonad: Info
web applications. Master'
Science, the University of

[18] Polytechnic Institute
- cybersecurity competiti
http://www.poly.edu/csa

[19] Prechelt, L. Plat_forms
comparison by an explora
emergent platform proper
Software Engineering 37,

[20] PostgreSQL: The world's
database. http://www.po

lems. 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

[21] Ruef, A., Hicks, M., Pa
Plane, J., and Mardzie
and comparing developm
[22] Saltzer, J. H., and Sch
information in computer
63, 9 (1975), 1278–1308.
[23] Scandariato, R., Wald
analysis versus penetratio
experiment. In *IEEE Int
Reliability Engineering (

[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, Predicting vulnerable com mining. In *IEEE Interna Reliability Engineering* (2

[28] Yang, J., Ryu, D., and prediction accuracy with measures. In *Internationa Smart Computing (BigC*

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