Applicability of the Botball Double Elimination Skill Set to the Real World
(Or: Why Making Robots Sabotage Each Other Is Actually Constructive)
Jeremy Rand
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy@veclabs.net

Applicability of the Botball Double Elimination Skill Set to the Real World (Or: Why Making Robots Sabotage Each Other Is Actually Constructive)

1 Introduction

The existence of head-to-head rounds in Botball, and the strategy choices that teams make for such rounds, have become a somewhat polarizing topic in the Botball scene. The primary arguments which I've personally heard regarding this topic tend to involve a sense of fair play, and making students satisfied with their results. In this paper, I will attempt to approach the issue from a different angle: I argue that having a significant level of interaction between teams in Double Elimination (DE) rounds teaches useful (actually vital) skills which are relevant across the programming workforce, particularly in the area of computer security. Specifically, being able to predict, find, and fix exploitable bugs, both in your software and other people's software, is the main skill which keeps critical software secure.

2 Who Are You?

I'm a former Botballer from Norman Advanced Robotics in Norman, OK (class of 2011); I co-led Team SNARC to win first place in DE at KIPR Open 2013 (we used a fairly aggressive manner of sabotage involving a large umbrella). I currently mentor three middle-school Botball teams in Norman (Alcott, Whittier Boys, and Whittier Girls), and as a result I'm exposed to a wide range of opinions on DE strategy. I also have a strong interest in hacking, reverse-engineering, and security. In these areas, I've published GCER papers on hacking various aspects of the XBC, CBC, Link, AR.Drone, and Create; I hacked an offline 2-player Nintendo GameCube game (Sonic Adventure 2: Battle) to play online with unlimited simultaneous players (via reverse-engineering); I hacked another GameCube game (Starfox Assault) to play in virtual reality with the Oculus Rift (also via reverse-engineering), and I'm a core developer at the Namecoin Project, which creates decentralized replacements for the DNS and TLS protocols which most computers rely on for network security. I generally credit Botball for both my interest and skill set in these topics.

As a result of this experience, I care (and know something) about Botball, STEM education, and their applicability to non-Botball situations.

3 How DE Scoring Typically Works

As my former teammates at NAR, Daniel Goree and Rhea Kickham, have stated [1], DE rewards teams for scoring *more* points rather than *the most* points. If you can prevent an opponent from

scoring more than 10 points, you're perfectly safe scoring only 15 points yourself. Usually speed and resistance to attack are the major factors in success, rather than a long program that scores a lot of points. Margin of victory is completely irrelevant in DE scoring. Needless to say, this encourages strategies which innovate by sabotaging an opponent's ability to score points, and by resisting attempts by opponents to do the same. Whether this is good or bad is the subject of some controversy.

4 Finding Exploits Is an Important Skill

In real-world programming, it is highly unlikely that you will produce bug-free code without asking other people to beta test. Even after the formal beta testing period is over, third parties will continue to find bugs. This can be a serious problem if your code is running in a mission-critical system.

For example, in April 2014, a critical vulnerability was found in the OpenSSL library which is used by about 2/3 of HTTPS websites (as well as a large number of other Internet services). The vulnerability, called Heartbleed, allowed anyone on one end of a secure connection to read arbitrary memory contents from the other end of the connection, including passwords, private keys, and other confidential materials [2]. The bug was introduced in December 2011, meaning that critical infrastructure was exposed for years. Even more troubling, evidence suggests that the Heartbleed attack was used in the wild in November 2013 [3].

How did this happen? Lack of code review. The OpenSSL project has been subject to much less security auditing than would be warranted for such critical infrastructure. Eventually, the bug was detected by security researchers who were actively searching for ways to break the security of OpenSSL. We need many more people doing this kind of research, and we need it yesterday.

5 DE Incentivises Finding Exploits

In DE, teams are rewarded for finding ways that opponent strategies might be exploited to prevent the opponent from succeeding at scoring the intended points. While it might seem that driving a Create in the way of a choke point has little to do with sending specially crafted TLS handshake packets, they actually have a similar underlying thought process: a programmer has to think, “How could the code's assumptions here go wrong?” Getting students to actively evaluate the validity of their assumptions, and other people's assumptions, in the face of attack prepares them for the real world of programming, where a failure of their assumptions could have catastrophic consequences.

6 Proactively Patching Exploits Reduces Accidental Failures

Even if you're not concerned about deliberate attacks, patching exploits can reduce accidental failures too. In my work with the Namecoin Project [4], we have a tool called NMControl that parses data specifying which websites map to which IP addresses. The specification for that data assumes that the data is ASCII-encoded (i.e. no non-English characters), but this assumption is not enforced anywhere. We figured that if a non-English character appeared, it would just break the website that put it there, meaning there would be no problem.

Then, we implemented some experimental code that processed every website in the database

proactively to reduce lookup time. And we noticed that it was crashing. Eventually we figured out that this was because there's a specific non-English character which crashes NMControl when it's accessed, and some website had erroneously placed that character in their data. We quickly released a patch, and no trouble resulted. But consider what would have happened if we hadn't tested and found this bug. Most likely it would have failed in the wild. It *probably* wouldn't be due to a malicious attack (although it could be), but an accidental failure would still be pretty bad if it occurred on critical infrastructure.

7 Learning Early Is Important

Students need to understand the value of questioning failure assumptions as soon as (preferably before) they are just competent enough at coding to be dangerous. A quick story: Zhou Tong was a high school student who built his own margin trading platform (which was often used by customers as a bank) for the Bitcoin digital currency [5]. (Yes, in the unregulated world of Bitcoin, a high schooler can build his own bank.) He didn't have any experience with security, but he was smart enough (and good enough at Googling) to make something that *looked* like it worked. He didn't actively try to break his platform, and he didn't invite people to try to break it in a controlled setting. In short, he was just competent enough at coding to be dangerous. His service was robbed, and his customers lost most of their money. [6]

Practicing breaking and fixing your code is something that you need to be taught immediately after you start coding, if you want to be good at it. As an analogy, most programmers start coding in college. This may sound alien, since the majority of readers of this paper probably started coding in middle school or earlier. But most programmers start in college. It turns out that learning a certain way of thinking about coding is much more effective if you practice it early. Take functional programming as an example. In functional programming, you don't have loops, you only have recursion. Why is that good? Mathematically, if your code has no loops, you can run your code through a math formula and you can prove whether your code has certain types of bugs. So you don't need to test your code over and over and hope you caught all the bugs, you simply do some math and you know for sure that your code is bug-free (to a certain extent). The problem is that code in functional languages feels different from C code; the thought process is different when you're coding. Parallel programming is similar. In parallel programming, you have many different CPU's all working on different parts of the same problem simultaneously. It's much more efficient than standard programming – in fact, it's necessary if you want to do certain kinds of very complex computations in reasonable time. But the thought process in writing parallel code is different from coding in C.

Dr. Rex Page at OU has discovered that if you teach functional programming early, the semester after students start coding for the first time, the students are much better able to do functional programming later [7]. That's what OU does now. I've heard similar results about parallel programming. And the same thing is true about finding ways your code can fail, and how those failures can be fixed. It's a thought process, which is different from just writing C code until you get the right result a few times, and it's much easier to get good at it if you start early.

8 Gamification Enhances Education

Many software companies offer bug bounties, particularly for critical code, because they realize

that motivating people to break their code is a way to prevent it from failing in dangerous situations. But this has limits, because breaking code is hard, and not necessarily always self-rewarding. So the Hack a Server Project [8], who had previously worked on bug bounty infrastructure, launched a new online game for programmers called Capture the Flag 365 [9], where you get points for 2 things: breaking other people's code, and fixing your code when other people break it. CTF365 is marketed as a way for programming students to learn important skills in a fun environment, so that they don't have to learn it the hard way (like Zhou Tong did). Just like CTF365, Botball is a fun environment to get good at these skills, so students don't have to learn this the hard way when they're in the workforce.

9 Learning to Identify Unworkable Approaches

Sometimes an exploit is so ingrained in how a system works that the system must be completely rebuilt to fix the exploit. That happens in Botball; for example, your robot must take 10 seconds to drive to an open scoring object, but other bots can get a giant arm there in 5 seconds. Probably the best instance of this is the 2006 Oklahoma Botball Tournament, where Norman High School (now part of Norman Advanced) was able to obtain the main scoring objects without driving (taking around 2-3 seconds to obtain control of the scoring objects), which completely broke any driving-based strategy. As a result, the only viable strategies for winning the 2006 Oklahoma DE tournament were non-driving strategies. A driving-based strategy was irreparably broken; and would have to be abandoned. Norman High won the Oklahoma regional.

The possibility of inherently broken designs also happens in real world programming. Your goal in Botball is to learn the skills so that if it happens to you in the real world, you can quickly notice the problem before you've invested months or years in the project, or worse, someone else notices it after you've released a product.

10 Learning to Identify Near-Impossible Problems

Sometimes an environment is so hostile that a huge amount of the solution space is irreparably vulnerable. Botball 2009 Oklahoma regionals were like this, where Whittier Boys' strategy of causing havoc and scoring 10 points was the only reliable winner. KIPR Open 2012 was similar, where any 2 teams using the AR.Drone would probably crash into each other, and the winning team (Lockheed Martin) didn't use an aerial robot.

This happens in the real world too. An example is social networking privacy. All centralized social networking websites require a funding source, and since users are unlikely to pay for such service, advertisers become the primary stakeholders. This harms privacy, but the rest of the solution space is unworkable. The only way to overcome this barren solution space is to eliminate the centralized service completely, as projects such as Friendica [10] attempt. Anyone mentally stuck inside the “box” of centralized providers would never come to a satisfactory solution.

Being able to identify these situations is very important, and coping with them is important too. In the 2009 Oklahoma regional season, Norman Advanced had thought of a similar strategy to Whittier Boys', but decided against it because it wouldn't be as fun to build and code as a giant arm. In retrospect, we absolutely should have done what Whittier Boys did. The most fun-to-make solution is not always the best.

11 Peer Review, Open Source Designs, and Security by Obscurity

In Botball, you can't keep a design secret forever. If you use it at a regional, everyone will probably see it before GCER (in part thanks to Botball Live). And even within GCER, if you bring a robot to a practice table, chances are many teams will take notice before they play you in DE.

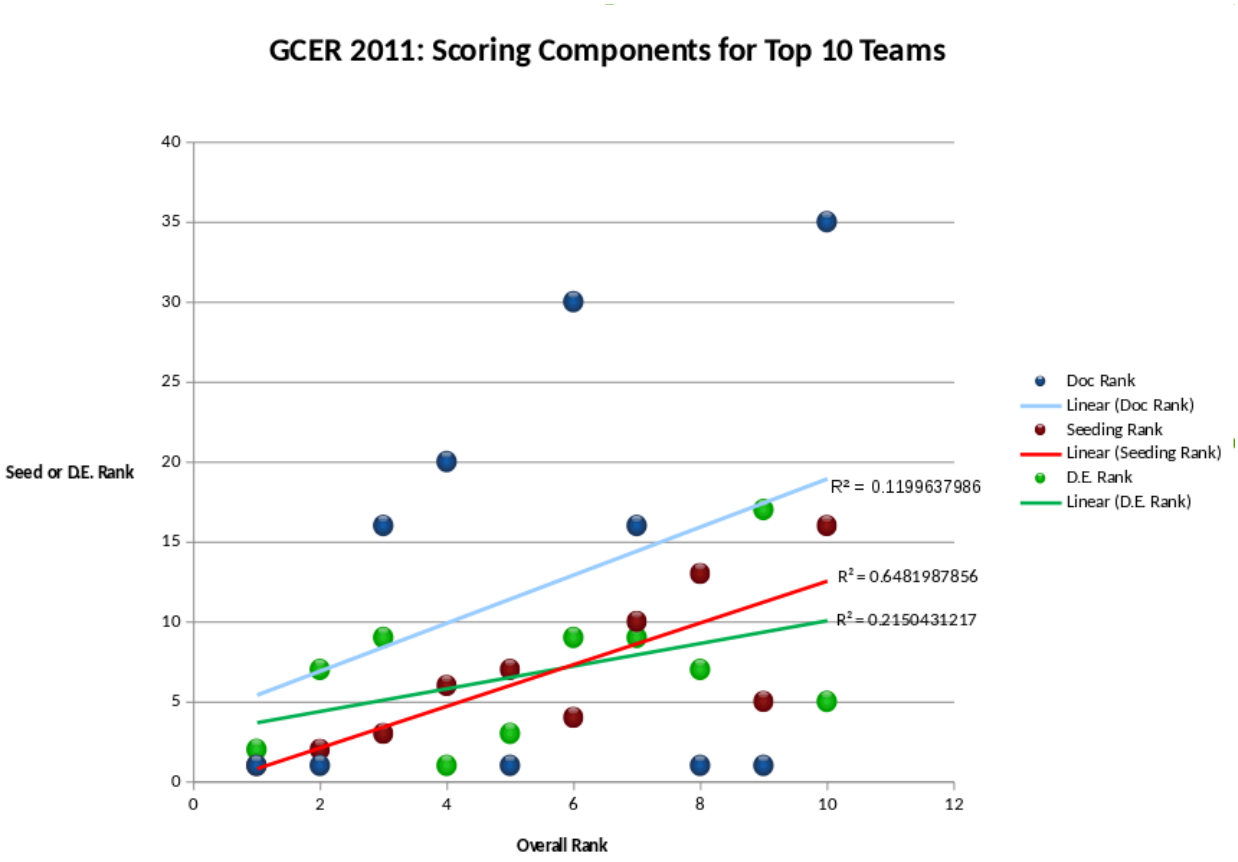
In 2006, Southwest Covenant (also in the Oklahoma region) knew this, and took full advantage. They had obtained videos of Norman High's non-driving robot at the Oklahoma regionals, and based their International Tournament strategy around beating this robot. They found a method which also used no driving, but only took around 1-1.5 seconds. Southwest Covenant's method scored fewer points, but was highly effective against Norman High (and slower strategies). By chance, Norman North High School (now also part of Norman Advanced) had a similar strategy as Southwest Covenant. Southwest Covenant ended up winning DE, and Norman North was the only team to tie the undefeated Southwest Covenant.

This mimics the professional process of peer review and open-source designs. Programmers should expect their code to be subjected to intense scrutiny – and only the code that can survive a thorough review should be considered secure.

However, there is something to be said for security by obscurity. Since Botball teams typically keep their strategies secret when feasible, there is not as much opportunity for opponents to review their strategies and find weaknesses. Security by obscurity, while not a substitute for the security that comes with peer review, is still widely used in industry.

12 Impact of DE on Overall Scoring

While Seeding, DE, and Documentation are widely assumed to have equal roles in Botball, the reality is quite different. A highly extreme example was 2011, most likely due to the exponential scoring. Below is a chart of the top 10 overall teams' ranks in the three categories.



As the graph shows, Seeding rank determined 64.8% of the Overall rank. DE was only 21.5%. Documentation trailed last with a mere 12.0%.

Is this good or bad? On the good side, it improves the determinism (and therefore competitive fairness) of the results, since Seeding has less randomness. On the other hand, DE has some major benefits, as detailed in this paper. In 2011, DE was of so little weight that the overall winner was known before the final bracket of DE had even begun – no team could beat Hanalani overall regardless of how the final bracket played out. At the 2011 Feedback Session, a Botballer pointed this out, and said that this made the DE final bracket a lot less motivating than it otherwise would be.

KIPR appears to have taken this criticism seriously, and I personally applaud them for doing so. DE is important, and it needs to matter.

13 Conclusion

DE plays an important role in teaching vital skills, which more than makes up for its randomness. DE's detractors should seriously take this into account. Some people might say, “If you're smart, you can score points without messing up the other team.” I say, “If you're truly smart, you can predict how other teams will mess up your robot and use that knowledge to score anyway, while figuring out how other teams can be messed up in ways that they can't prevent.”

14 References

[1] Daniel Goree, Rhea Kickham. *Considerations for Strategy Development*. Proceedings of the 2012 Global Conference on Educational Robotics. http://files.kipr.org/gcer/2012/proceedings/_Goree-Kickham_Considerations_Norman.pdf

[2] Codenomicon. *Heartbleed Bug*. http://heartbleed.com/

[3] Peter Eckersley. *Wild at Heart: Were Intelligence Agencies Using Heartbleed in November 2013?* https://www.eff.org/deeplinks/2014/04/wild-heart-were-intelligence-agencies-using-heartbleed-november-2013

[4] *Namecoin*. http://namecoin.info/

[5] The Bitcoin Foundation. *Bitcoin – Open source P2P money*. https://bitcoin.org/en/

[6] Bitcoin Forum. *[Emergency ANN] Bitcoinica site is taken offline for security investigation.* https://bitcointalk.org/index.php?topic=81045.0

[7] Rex L. Page. *Curriculum Vita*. http://www.cs.ou.edu/~rlpage/cvrlpage.html

[8] *Hack a Server*. http://hackaserver.com/

[9] *Capture the Flag 365*. http://ctf365.com/

[10] Mike Macgirvin. *Friendica*. http://friendica.com/