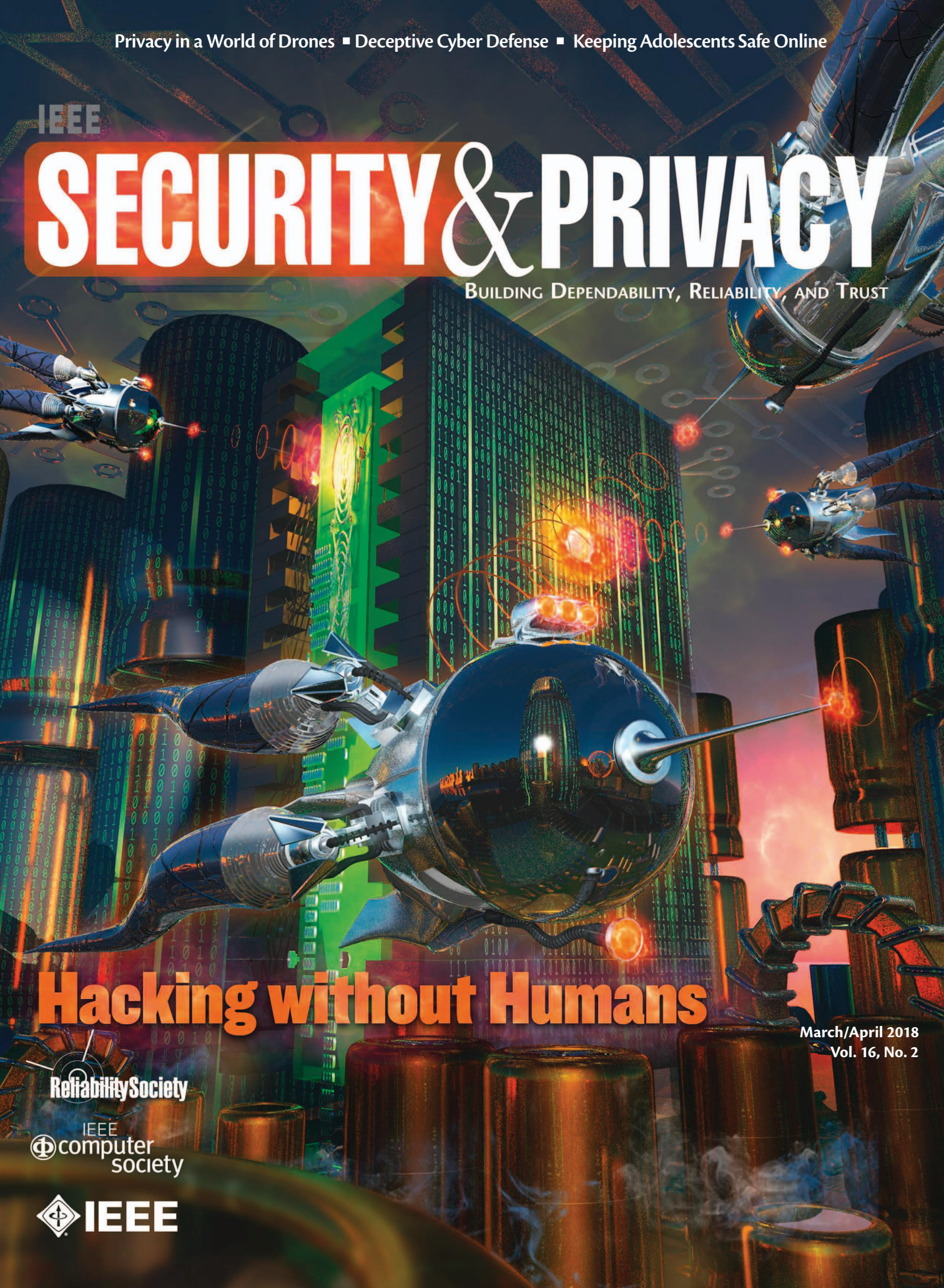
Privacy in a World of Drones ■ Deceptive Cyber Defense ■ Keeping Adolescents Safe Online

IEEE

SECURITY & PRIVACY

BUILDING DEPENDABILITY, RELIABILITY, AND TRUST



Hacking without Humans

March/April 2018
Vol. 16, No. 2

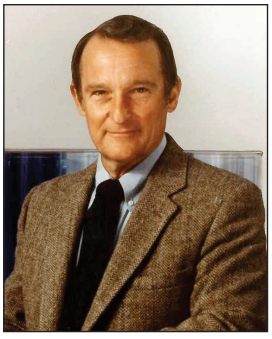
Reliability Society
IEEE computer society
IEEE

Recognizing Excellence in High Performance Computing

Nominations are Solicited for the

SEYMOUR CRAY SIDNEY FERNBACH & KEN KENNEDY AWARDS

SEYMOUR CRAY COMPUTER ENGINEERING AWARD



Established in late 1997 in memory of Seymour Cray, the Seymour Cray Award is awarded to recognize innovative contributions to high performance computing systems that best exemplify the creative spirit demonstrated by Seymour Cray. The award consists of a crystal memento and honorarium of US$10,000. **This award requires 3 endorsements.**

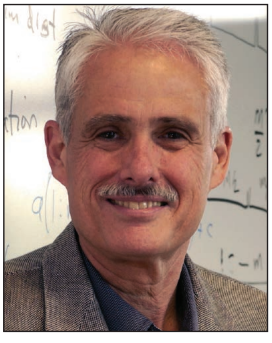
Sponsored by: IEEE computer society

SIDNEY FERNBACH MEMORIAL AWARD

Established in 1992 by the Board of Governors of the IEEE Computer Society. It honors the memory of the late Dr. Sidney Fernbach, one of the pioneers on the development and application of high performance computers for the solution of large computational problems. The award, which consists of a certificate and a US$2,000 honorarium, is presented annually to an individual for "an outstanding contribution in the application of high performance computers using innova-tive approaches." **This award requires 3 endorsements.**

Sponsored by: IEEE computer society

ACM/IEEE-CS KEN KENNEDY AWARD



Established in memory of Ken Kennedy, the founder of Rice University's nationally ranked computer science program and one of the world's foremost experts on high-performance computing. A certificate and US$5,000 honorarium are awarded jointly by the ACM and the IEEE Computer Society for outstanding contributions to programmability or productivity in high performance computing together with significant community service or mentoring contributions. **This award requires 2 endorsements.**

Cosponsored by: IEEE computer society, acm Association for Computing Machinery

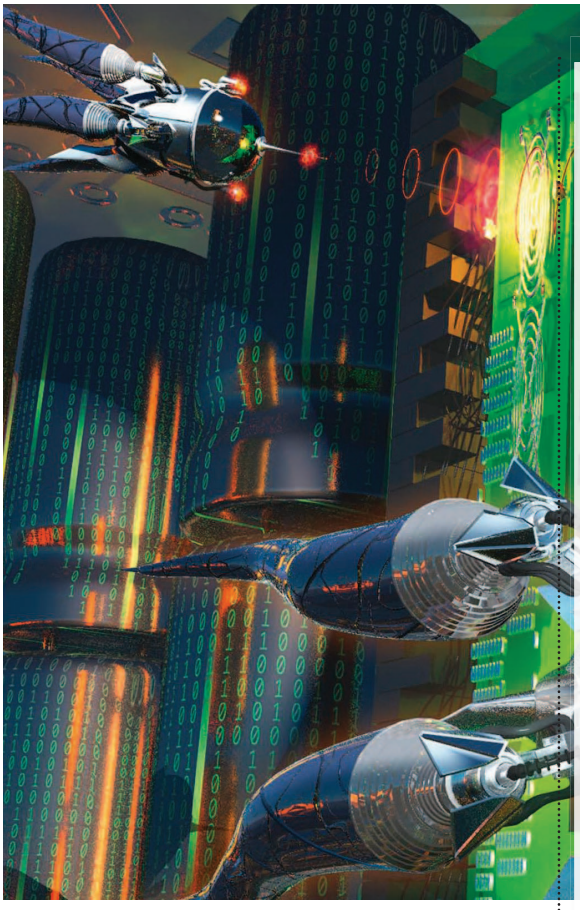
Deadline: 1 July 2018

All nomination details available at http://awards.computer.org

IEEE

March/April 2018
Vol. 16, No. 2

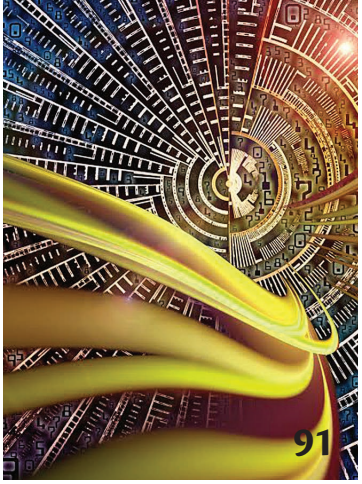
Contents



Cover art by Barry Downard, www.debutart.com

Hacking without Humans

The Cyber Grand Challenge presented an opportunity to advance the state of the art and science in autonomous reasoning in software security. The culmination of this contest presented seven fully autonomous systems vying against each other hacking and defending previously unseen computer software, ultimately making sense of zero-day attacks, entirely without any human assistance. This collection of works presents many aspects of autonomous software security from differing perspectives. The authors detail solution approaches as well as evidence of the challenges that remain.



Also in This Issue

Column

Departments

Also in This Issue

Postmaster: Send undelivered copies and address changes to *IEEE Security & Privacy*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals postage rate paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8. Printed in the USA. **Circulation:** *IEEE Security & Privacy* (ISSN 1540-7993) is published bimonthly by the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, phone +1 714 821 8380; IEEE Computer Society Headquarters, 2001 L St., Ste. 700, Washington, D.C. 20036. Subscribe to *IEEE Security & Privacy* by visiting www.computer.org/security. *IEEE Security & Privacy* is copublished by the IEEE Computer and Reliability Societies. For more information on computing topics, visit the IEEE Computer Society Digital Library at www.computer.org/csdl.

Introduction from the New EIC

David Nicol
Editor in Chief

I was honored to be asked to be nominated to the position of editor in chief (EIC) of *IEEE Security & Privacy*, and even more honored to be chosen. I come to the job with six past years of EIC experience, almost two decades ago serving *ACM Transactions on Modeling and Computer Simulation*. This very fact hints that my path to security was out of the norm. From my 1985 PhD on to 1999, I worked primarily on parallel discrete-event simulation. But in 1999, while a professor of computer science at Dartmouth, along with George Cybenko (a former *IEEE S&P* EIC!), I was pulled into an overture by Senator Gregg of New Hampshire to establish at Dartmouth what became the Institute for Security Technology Studies (ISTS). At the time, Dartmouth had no security expertise, but Cybenko and I worked at applying our own research areas to security and encouraged others at Dartmouth to do the same. For me, that meant bringing a systems modeling point of view to the question of assessing how well a system design is able to limit access to critical data and other resources needing protection. In 2002, I was recruited by the University of Illinois at Urbana-Champaign's (UIUC's) Electrical and Computer Engineering Department, which I joined in 2003. About the time of my arrival, Bill Sanders became founding director of the Information Trust Institute (ITI; www.iti.illinois.edu) as a result of responding to an NSF CFP for Engineering Research Centers (ERC) and leading a bid representing UIUC faculty with common interests in security, reliability, and distributed systems. The ERC bid was not successful, but Sanders, other faculty, and most importantly, UIUC administration realized there was a corpus of expertise in a vital research area. UIUC obtained seed funding from the State of Illinois to launch ITI. My research in support of ITI really defines where I'm coming from with respect to security.

ITI's trajectory was shaped by a successful bid for an NSF-funded center focused on cybersecurity in the electric power grid. Called Trustworthy Cyber Infrastructure for Power (TCIP), the center proposed collaborative research among computer scientists/engineers and electric power engineers from four universities to address real security problems encountered in the grid. TCIP placed high emphasis on interaction with vendors and utilities serving the power grid. Successes with TCIP led to the Department of Energy (DOE) taking this research group to the next level in 2009 with the Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) project, and the trust ITI had established with the DOE and industry led to our participation in various projects initiated by industry and funded by the DOE. In this same time period, I was active representing UIUC in the Institute for Information Infrastructure Protection (I3P) Consortium, where I worked on several projects aimed at improving cybersecurity in the oil and gas sector.

In 2011, I became director of ITI, with the responsibility of helping to start and manage center-scale activities focused on information trust. In 2015, ITI led a response involving 12 universities and national labs to the DOE's CFP for an academic center studying cyberresilience in energy delivery systems; this program now funds our Cyber Resilience for Energy Delivery Systems Consortium (CREDC; see www.cred-c.org).

As a result of work initially funded by I3P, then by NSF and DOE (under TCIPG), and finally DHS, Sanders, myself, and others started a company that licenses software used primarily in the conduct of North American Energy Reliability Corporation (NERC) Critical Infrastructure Protection (CIP) compliance audits. Power companies that generate or transmit sufficiently large amounts of power are periodically audited by NERC for a variety of things; our technology supports the portion of an audit that determines whether a so-called "critical asset" is protected from outside access and that all outside accesses to it are known,

1540-7993/18/$33.00 © 2018 IEEE Copublished by the IEEE Computer and Reliability Societies March/April 2018

documented, and justified. Our solution to this problem is a direct reflection of past work Sanders and I have done on system modeling, formal methods, and algorithms.

I share this all to reveal the sources of my perspectives and the emphases I intend to bring to this job. In my application for the EIC position, I shared with the recruiting committee the following list of topics I think are important, some of which I hope to explore in *IEEE S&P*:

- *Usable security.* Why is the current state of the practice so awful and what can be done about it? How can one distill complex computer-oriented policy to a level where a user has choices, understands what those choices are, and understands the ramifications of the choices made?
- *Investment in security.* Promoting cybersecurity as fire insurance against a low-probability Very Bad Day has proven to be a losing proposition. What sort of products can be viewed as adding positive service or return every day, while increasing cybersecurity at the same time?
- *Blockchain.* This technology has captured the imaginations of many for applications well beyond cryptocurrency. What's the real deal here? What advantages and disadvantages does it bring, and in what contexts?
- *Malware.* The demonstrated capability of malware in general and ransomware in particular to move laterally through a network means that it takes only one employee in an organization to admit havoc by clicking on a malicious attachment. The best intentioned employee can be served up malvertising because an organization's patching policy leaves a vulnerable browser or plug-in in place. Given that training cannot stop someone from taking an action that has high consequences, why is this still a problem? Why can't (or don't) mailers and operating systems protect us from ourselves? What technologies exist or are on the horizon to do this?
- *Policy.* Protection of privacy is an area that requires policy and law, along with technology, to support protection and detect violations of that policy and law. This is complicated. *IEEE S&P* can be a platform for asking (and answering) what privacy protections we have or can claim in social media, in online interactions, or even presence in public spaces (with respect to video surveillance).
- *Cyberinsurance.* Insurers see the potential for a huge, as yet largely untapped market. What's the current basis for premiums? What is covered? Where's the sweet spot among assessment technology, coverage, price, and willingness to pay?
- *Standards.* National and industry standards a computer system or network ought to meet continue to grow and be refined, but are seen by nonexperts as overwhelming and difficult to understand; consequently, they have no impact on businesses and organizations without the expertise or budget to improve cybersecurity in the most effective ways. How can this gap be bridged?
- *Effective compliance.* As cybersecurity standards evolve, there may be increased regulation, requirements from business partners, and/or legal expectation for companies to demonstrate that their systems meet standards. Where is there a sweet spot (if any) in terms of standards that balance the cost of demonstrating compliance with the effectiveness of the controls selected?
- *Risk assessment of cyber-physical systems.* Computer systems control virtually all cyber-physical critical infrastructures. The assessment of the threat (and cost) of damage to the infrastructure through the cyber component is critical to understanding how to prioritize investments in cybersecurity. What is the state of the practice? What are the standards that must be met by law? What technologies and practices are needed still to improve the state of the art in this area?
- *Machine learning and deep learning.* ML/DL has captured the imagination of many, and a natural question asks about application in cybersecurity. What are those applications and what promise does ML/DL have? What limitations and risks are there to moving in such a direction within a cybersecurity context?
- *Supply chain issues.* Computer systems are complex ensembles with hardware and software components drawn from all over the world. What can and should businesses be doing to develop their own trust, as well as consumer trust, in those components? What technical and legal means exist to develop that trust?
- *Cloud-based security services.* Companies are under pressure to improve their computer security posture but, without the in-house resources to do so, might look to cloud-based services. How do they work and what benefit do they provide? Are there privacy issues? What's on the horizon technology-wise?

I'm very much looking forward to working with the *IEEE S&P* editorial board and the IEEE Computer Society to lead exploration of these issues. But most important, I appeal to you the *IEEE S&P* readership to participate. Please consider writing articles or opinion pieces that address any of these topics in a way that informs and generates discussion, and please consider volunteering to lead the development of special issues on these or other practical problems we face in securing our computer systems. ■

IEEE
SECURITY & PRIVACY
www.computer.org/security

EDITORIAL OFFICE
IEEE Security & Privacy
c/o IEEE Computer Society Publications Office
10662 Los Vaqueros Circle, Los Alamitos, CA 90720 USA
Phone | +1 714 821-8380; Fax | +1 714 821-4010

Editor: Gary McGraw, gem@synopsys.com

Silver Bullet Talks with Craig Froelich

Gary McGraw | Synopsys

Hear the full podcast and find show links, notes, and an online discussion at www.cigital.com/silverbullet.

Craig Froelich is the chief information security officer (CISO) of Bank of America. He leads the global InfoSec team responsible for security strategy, policy, and programs. Prior to moving to Bank of America through acquisition, he was responsible for countrywide cybersecurity technology, networks, crisis management, and security operation. His background includes more than a decade of experience in his early career spanning product management and application development for software and hardware companies.

Thanks for joining us.
I really do appreciate it. I'm excited to be here.

Do did you set out with a goal of becoming a CISO; if not, how did your career progress?
No. In fact, I never considered CISO as a destination or even a stop along my career. When I first started thinking about what it was like to work in a technology organization, I had always thought about doing something that was really more about either application development or infrastructure, and inevitably, I realized when I was doing appdev or infrastructure work, I was really bad at it.

I always found it interesting when I would sit down and chat with the information security team at whichever company I was working with at the time. I always had a really interesting and good conversation, and I realized that I had passionate interest, so I just started heading down that path. But it wasn't necessarily something that I carved out and said, "I have to be a CISO." In fact, there wasn't even that title when I first started going down this path. It was really more of an interest, and it's something that I still wake up really excited about, because each day brings new challenges we get to figure out how to deal with.

I have been working on a project called the CISO Report. You can think of it as BSIM for CISOs; it's very much a data-driven project with a lot of in-person interviews. We interviewed 25 CISOs from various companies. So, my questions for you are: Why did you agree to help with this project, did you like the results, and how do you think the work can be used by other people?
I'm a big believer in benchmarks and being able to have a comparison to understand how we are doing as an organization, as a team, as individuals. This is such a nascent space. Information security and the information security teams within these companies have changed a lot over the last 10, 15 years, and we still have a lot of room for improvement. If there's a way for us to do a comparison against folks who are doing this well, all the better.

So, I was really excited that Bank of America was one of the firms that was asked to participate, and I'm hopeful that more of these types of programs and benchmarks and analyses will help people understand what the definition of *good* is. How do they know what it means to be a good—in this particular case—CISO, or what it means to run a good information security team?

By doing this, we can go back to all the people that we work for, that we are responsible for taking care of, the board members and the management team—go to them hand on heart and say, "Here's where we stand. Here's what we're doing, and here's what you should be expecting from us."

If we look at what the best people are doing and how people are approaching this job, and build a benchmark, we can all improve in a measureable way, I suppose.

Copublished by the IEEE Computer and Reliability Societies

Yeah, that's right. CISO is a pretty generic title, and a lot of people make assumptions as to what that role is. But what was really interesting about the work that you did was show that a CISO is not just a title, but the person combined with the company that they work for makes a magic combination of either a success or not a success. And that it really is a little bit more scientific than not.

What are your thoughts about overlap of CISO responsibilities? Are we doomed to a generic framework like the one that we came up with, or can we get more specific?

I think we can get more specific, but it's going to require us to iterate our way through. Many of us have different responsibilities, different scopes, we sit in different parts of the organization, we're funded differently. And I think it goes back to the difference between how organizations view the importance of information security.

Many companies deal with the information security challenge as a function of compliance. They'll go through and make sure they comply with all of the laws, rules, and regulations and check that box. But a lot of companies are increasingly looking at it as something that's more than just compliance—they have to be committed to doing it. The commitment starts from the top down; it is an obligation of everybody who works in the organization.

For those companies that have transitioned from compliance to commitment, the role of a CISO is different. And so, as we iterate our way through these generic frameworks and move to something that's a little bit more specific, it'll allow us to decompose the organizations that have made the transition and measure them accordingly.

In the CISO project, we identified four major groups, or tribes, of CISOs: security as cost center, security as compliance, security as technology, and security as enabler. What did it take to become a "security as enabler" CISO, and how did you change from a highly technical guy into a business person?

In any relationship, it does come down to two parties. So the company, Bank of America, was already looking for somebody who could be an enabler. They didn't want to have somebody who fell into the other tribes. And that's really more of a statement of my predecessors who came in and laid a strong foundation and helped them understand what it took for this organization to be successful. And that looked and smelled an awful lot like an enabler.

In terms of my role in that, you're right. I came from a technology background and I'm really comfortable with technology. But it's not enough to be able to sit in front of the board and to talk about all the technical things that we're proud of, because when it comes down to it, the board, the management team, they need to know beyond technology that, when the chips are down, the person sitting in front of them is going to be able to lead them through a potential crisis.

And that confidence is something that is built up; it's not just about technical aptitude. It's about leadership. It's about knowledge of the business. It's about knowing the difference between which things are critical and which are not. I was really fortunate having been a part of Bank of America for a long time. They gave me an opportunity to work in a lot of different parts of the company. And through those different roles, I learned a lot about how the place is wired and what is important. So when I sit in this seat today, I have a great deal of empathy for all the people in the organizations that I've worked for because I've done a lot of those jobs and I can take what they do and explain it with confidence and, like I said, with the empathy that it deserves.

But at the same time, because I understand the company, I can also understand what is important to the management team and where to be able to draw that line around how to protect the firm without strangling it because we are being too restrictive and putting in place the wrong type of capabilities and controls.

You've built a remarkable team of talented people in your organization who come from wildly different backgrounds. How did you do that, and how do all these incredibly successful people on totally different dimensions work as a team?

It's a very intentional strategy. When I sit down with the board of directors, my strategy in many cases can

About Craig Froelich

Craig Froelich is the CISO of Bank of America. He leads the global InfoSec team responsible for security strategy, policy, and programs. Prior to moving to Bank of America through acquisition, he was responsible for countrywide cybersecurity technology, networks, crisis management, and security operation. His background includes more than a decade of experience in his early career spanning product management and application development for software and hardware companies. Froelich serves on the board of the Financial Services Information Sharing and Analysis Center and the executive committee of bits. He describes himself on Twitter as a SoCal dude learning to be a Southern gentleman. He lives in North Carolina with his family.

be summarized in four words: best tech, best talent. And if you decompose that statement, you have to have best talent in order to have best technology because it's the people that make the technology go. So being able to make sure that we have an outstanding, talented workforce is something that I spend almost all my time on. Today, we have a team of about 2,400 information security professionals dedicated to defending the firm.

That diversity of background and of experience is important to me, because if we're trying to anticipate the actions of an adversary, we can't just have a bunch of people who look and act and sound the same. We need to be able to make sure that we get people from all walks of life, different backgrounds, different experiences, different races, different ethnicities, different genders all under one roof. And it's important that we give them a really hard problem and the ability to solve it.

I've found, more often than not, that's really what a lot of people are motivated by. We do everything necessary to not only protect our clients and customers but also make sure that we have a sound and safe financial services ecosystem—that's a really hard problem. And Bank of America gives us the budget and the ability to spend that money to ensure that we have the ability to solve that problem.

The CEO often talks publicly about how the information security function within the organization is unconstrained. Unconstrained budget, unconstrained in terms of our ability to effect change. And that sends a really powerful message not only to people's accountability within the company. Every employee understands why we do that, but it's a great way to ring the dinner bell for anybody who is interested in thinking about information security as a career or is already doing that and wants to come to an organization and learn a lot.

Some organizations are constrained in their security approach by budget, and others are not. But all organizations certainly have a maximum capacity for properly absorbing security. And the field changes a lot. So how do you figure out how much to do at once, and what not to do now?

It's not enough just to have the money; you have to be able to spend it. The ability to effect change within the organization is necessary when you have that kind of commitment from the management team. So yes, we could spend it and potentially go so fast that we're introducing risk, but that goes back to the need to be able to balance business expectations with the program that you're running as a security professional.

In most cases, companies can move faster than they think. And finding that maximal velocity without being unsafe is the dance that I think any CISO has to face.

Let's talk about my favorite subject: software security. Do you think we've made progress in applied security engineering?

We've made some progress, but you only have to look back at some of the big headlines over the past 12 to 18 months to see that we have a lot of work to do. All the headlines have had one thing that is very consistent: it was a vulnerability that was either known or should have been known to management. Most often, those were software vulnerabilities, and so we're still seeing them too frequently. We're still seeing them with the potential for too much damage, and they're not something that is well-known across the tech community or the management team that's responsible for making sure that it can't be exploited.

So I think we've done a lot as an industry, but we have a long way to go from there.

I agree. The way I would put it is, we know what to do, and now we have to do it.

Yeah. Well said.

So, tell us what it's like to be an LA boy foist into the genteel Southern gentlemen culture in North Carolina.

Coming from LA to Charlotte—I come with a built in network folks. For my wife and kids, it was harder because they didn't know anybody here. But Charlotte as a city is a remarkably open community. Very few people in Charlotte are from Charlotte, so I think everybody has recollection of what it was like to move here and they remember that, but at the same time, they also pick up that Southern hospitality and open charm.

Very last question. What music is in heavy rotation in your life at the moment?

Right now, I've been doing a few things. So one, I love the new U2 album. That seems to be picking up a lot of rotation at the moment. And for some reason, I've been doing a whole South American, Central American thing. So I've got that in rotation. It just depends on the day, the mood.

The Silver Bullet Podcast with Gary McGraw is cosponsored by Cigital (part of Synopsys) and this magazine and is syndicated by SearchSecurity. ■

Gary McGraw is vice president of security technology at Synopsys. He's the author of *Software Security: Building Security In* (Addison-Wesley 2006) and eight other books. McGraw received a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him via garymcgraw.com.

IEEE computer society

PURPOSE: The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.
MEMBERSHIP: Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.
COMPUTER SOCIETY WEBSITE: www.computer.org
OMBUDSMAN: Direct unresolved complaints to ombudsman@computer.org.
CHAPTERS: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.
AVAILABLE INFORMATION: To check membership status, report an address change, or obtain more information on any of the following, email Customer Service at help@computer.org or call +1 714 821 8380 (international) or our toll-free number, +1 800 272 6657 (US):

- Membership applications
- Publications catalog
- Draft standards and order forms
- Technical committee list
- Technical committee application
- Chapter start-up procedures
- Student scholarship information
- Volunteer leaders/staff directory
- IEEE senior member grade application (requires 10 years practice and significant performance in five of those 10)

PUBLICATIONS AND ACTIVITIES
***Computer*:** The flagship publication of the IEEE Computer Society, *Computer*, publishes peer-reviewed technical content that covers all aspects of computer science, computer engineering, technology, and applications.
Periodicals: The society publishes 13 magazines, 19 transactions, and one letters. Refer to membership application or request information as noted above.
Conference Proceedings & Books: Conference Publishing Services publishes more than 275 titles every year.
Standards Working Groups: More than 150 groups produce IEEE standards used throughout the world.
Technical Committees: TCs provide professional interaction in more than 30 technical areas and directly influence computer engineering conferences and publications.
Conferences/Education: The society holds about 200 conferences each year and sponsors many educational activities, including computing science accreditation.
Certifications: The society offers two software developer credentials. For more information, visit www.computer.org/certification.

NEXT BOARD MEETING

7-8 June 2018, Phoenix, AZ, USA

EXECUTIVE COMMITTEE

President: Hironori Kasahara
President-Elect: Cecilia Metra; **Past President:** Jean-Luc Gaudiot; **First VP, Publication:** Gregory T. Byrd; **Second VP, Secretary:** Dennis J. Frailey; **VP, Member & Geographic Activities: Forrest Shull**; **VP, Professional & Educational Activities: Andy Chen**; **VP, Standards Activities:** Jon Rosdahl; **VP, Technical & Conference Activities:** Hausi Muller; **2018-2019 IEEE Division V Director: John Walz**; **2017-2018 IEEE Division VIII Director:** Dejan Milojicic; **2018 IEEE Division VIII Director-Elect:** Elizabeth L. Burd

BOARD OF GOVERNORS

Term Expiring 2018: Ann DeMarle, Sven Dietrich, Fred Douglis, Vladimir Getov, Bruce M. McMillin, Kunio Uchiyama, Stefano Zanero
Term Expiring 2019: Saurabh Bagchi, Leila DeFloriani, David S. Ebert, Jill I. Gostin, William Gropp, Sumi Helal, Avi Mendelson
Term Expiring 2020: Andy Chen, John D. Johnson, Sy-Yen Kuo, David Lomet, Dimitrios Serpanos, Forrest Shull, Hayato Yamana

EXECUTIVE STAFF

Executive Director: Angela R. Burgess
Director, Governance & Associate Executive Director: Anne Marie Kelly
Director, Finance & Accounting: Sunny Hwang
Director, Information Technology & Services: Sumit Kacker
Director, Membership Development: Eric Berkowitz
Director, Products & Services: Evan M. Butterfield

COMPUTER SOCIETY OFFICES

Washington, D.C.: 2001 L St., Ste. 700, Washington, D.C. 20036-4928
Phone: +1 202 371 0101 • **Fax:** +1 202 728 9614
Email: hq.ofc@computer.org
Los Alamitos: 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 **Phone:** +1 714 821 8380
Email: help@computer.org

MEMBERSHIP & PUBLICATION ORDERS
Phone: +1 800 272 6657 • **Fax:** +1 714 821 4641 • Email: help@computer.org
Asia/Pacific: Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan
Phone: +81 3 3408 3118 • **Fax:** +81 3 3408 3553
Email: tokyo.ofc@computer.org

IEEE BOARD OF DIRECTORS

President & CEO: James Jefferies
President-Elect: Jose M.F. Moura
Past President: Karen Bartleson
Secretary: William P. Walsh
Treasurer: Joseph V. Lillie
Director & President, IEEE-USA: Sandra "Candy" Robinson
Director & President, Standards Association: Forrest D. Wright
Director & VP, Educational Activities: Witold M. Kinsner
Director & VP, Membership and Geographic Activities: Martin Bastiaans
Director & VP, Publication Services and Products: Samir M. El-Ghazaly
Director & VP, Technical Activities: Susan "Kathy" Land
Director & Delegate Division V: John W. Walz
Director & Delegate Division VIII: Dejan Milojičić

IEEE

revised 7 Feb. 2018

Changing the Game of Software Security

Timothy Vidas | Secureworks
Per Larsen | Immunant
Hamed Okhravi | MIT Lincoln Laboratory
Ahmad-Reza Sadeghi | Technische Universität Darmstadt

DARPA's Grand Challenges are meant to invoke a type of innovation that is difficult to attain through traditional research avenues. Perhaps the most memorable, the 2004 effort toward self-driving vehicles, was simply dubbed the "Grand Challenge" at the time. Several such challenges have been designed and executed by DARPA since, each pushing the boundaries of science and technology. In computer and network security, a similar drive and innovation are present in a contest environment known colloquially as "capture the flag" or simply CTF.

The term "CTF" is borrowed from the physical game of capturing and defending literal flags. Today, the more apt analogy is likely the virtual variety found in first-person shooter video games. A computer security CTF often has digital flags, typically a sequence of secret bytes, that participants must defend and/or attack. Without delving into the details of the now rich and diverse community of such CTF contests, suffice it to say that the contests have grown in complexity and difficulty since the mid-1990s. Furthermore, the most difficult and well-regarded CTFs attract competitive teams that curate strategy and capability for years.

At its core, the Cyber Grand Challenge (CGC) was meant to discern whether an autonomous, purpose-built system could compete in the highest levels of computer security CTFs. Years of effort culminated in the summer of 2016 as the CGC Final Event (CFE) was held in conjunction with the DEF CON conference.

This issue of *IEEE Security & Privacy* explores several aspects of autonomy with respect to computer hacking from varying perspectives centered on the CGC. As such, it is worthwhile to introduce the CGC parlance used throughout this issue. The competitors in the CFE were autonomous machines, physical racks of high-performance computing gear, dubbed cyber reasoning systems (CRSs). Obviously humans designed and programmed the inner workings of

 Copublished by the IEEE Computer and Reliability Societies

each CRS, but at the CFE, humans were mere spectators. The "course" that every CRS had to "navigate" was in the form of novel, known-vulnerable software. These challenge sets (CS) were composed of challenge binaries (CB) that were uniformly distributed to each CRS, which in turn had to 1) determine the vulnerable conditions and 2) prove that the conditions existed on opponents while simultaneously thwarting such attempts by others.

Every few minutes, a new round would begin, meaning that CSs may be introduced or removed, and proofs of vulnerability (PoVs) could be launched several times against various opponents. Concomitant, each CRS could elect to mitigate vulnerabilities; however, CRS-fielded CBs (and network IDS signatures) were readily made available to opponents, mimicking some properties of real-world patching paradigms.

Many articles in this issue mention DECREE, a CGC-specific operating system interface specification. DECREE was created to narrow both the space in which the contestants competed and also the risk present from evaluating competitor-provided software. Unlike the hundreds of system calls present in modern operating systems, DECREE employs seven. The seven specific calls were meant to be expressive enough to model most memory-related vulnerability classes. The binary format for DECREE borrows heavily from the common Executable Linkable Format (ELF) file format, and the competition framework integrity team implemented DECREE on both 32-bit Linux and 64-bit FreeBSD.

In the end, seven diverse CRS finalists all successfully participated in the CFE. After 96 rounds, or just over nine hours, one emerged victorious (the winning team contributed the article on page 52 of this issue). Foremost, the CGC proved that a CRS could be built—that is, a computer could play in a CTF-style event, by itself. The CGC also provided a specification for an autonomous, brokered CTF that has already been reused in other events, as has the special binary specification for CBs. Such reuse and the various CGC-related corpora are giving researchers common ground on which to further advance that state of the art.

Much work remains, however. Most CRS creators will readily admit that the reasoning aspects of their CRS (for instance, game theory, artificial intelligence, and machine learning) were rudimentary. Indeed, as the CFE was the first contest of its kind, there was no historical record to guide training. Similarly, it is difficult to discreetly articulate individual advances in any particular component domain employed by a CRS. For instance, fuzzing technology materially advanced during the CGC timeframe, but would the same advancements have occurred absent the CGC? Perhaps the most telling metric, of the 82 CSs employed in CFE, vulnerabilities were only proven in 20 (that is, less than a quarter).

No automobiles completed DARPA's 2004 challenge course. Just months later, in 2005, not only was a victor declared, but 22 of 24 contestants successfully navigated the rural course. In 2007, six contestants similarly completed an urban course. Now, 13 years later, we are seeing fully autonomous vehicles navigate public roads alongside human drivers. In 2016, every contestant in the world's first autonomous computer hacking tournament demonstrated a level of proficiency in autonomous vulnerability discovery, proof, and mitigation. It makes one wonder what levels of autonomy will be achieved in software security in the coming decade. ■

Timothy Vidas is a senior distinguished engineer at Secureworks. Contact at tvidas@secureworks.com.

Per Larsen is currently the CEO of Immunant. Contact at perl@immunant.com.

Hamed Okhravi is a senior staff member at MIT Lincoln Laboratory. Contact at hamed.okhravi@ll.mit.edu.

Ahmad-Reza Sadeghi is a professor of computer science at Technische Universität Darmstadt. Contact at ahmad.sadeghi@trust.cased.de.

Mechanical Phish: Resilient Autonomous Hacking

Yan Shoshitaishvili | Arizona State University
Antonio Bianchi and Kevin Borgolte | University of California at Santa Barbara
Amat Cama | Independent Researcher
Jacopo Corbetta | Independent Researcher
Francesco Disperati | PayJunction
Audrey Dutcher | University of California at Santa Barbara
John Grosen | Massachusetts Institute of Technology
Paul Grosen, Aravind Machiry, and Chris Salls | University of California at Santa Barbara
Nick Stephens | Independent Researcher
Ruoyu Wang and Giovanni Vigna | University of California at Santa Barbara

Recently, the vulnerability analysis process has started to shift from human analysts to automated approaches. The DARPA Cyber Grand Challenge featured cyber reasoning systems, such as our Mechanical Phish, that analyze code to find vulnerabilities, generate exploits to prove the existence of these vulnerabilities, and patch the vulnerable software.

Our world is becoming increasingly connected, and the fantastical view of hackers, as portrayed by *Hackers* and other '90s-era movies, is starting to seem feasible, but with nation-states and criminal enterprises taking the place of Angelina Jolie and her crew. Because we have repeatedly demonstrated the lack of sufficient collective security experience (or sufficient interest in software security) to avoid widespread vulnerabilities, research has turned to the automatic discovery and repair of such flaws in deployed software.

One driver of this research direction is DARPA, who has a long track record of pushing for the automation of tasks traditionally (and imperfectly) handled by humans. DARPA bootstraps research areas through a time-proven method: explicit competitions. To advance autonomous security analysis, DARPA organized the Cyber Grand Challenge (CGC), a competition in which human teams built fully autonomous cyber reasoning systems (CRSs) that were pitted against one another in a contest to analyze, exploit, and patch binary software.

Like DARPA's earlier self-driving Grand Challenge, the CGC was a proxy for a realistic scenario. The first self-driving Grand Challenge was held in the desert, and the resulting prototypes would suffer driving in a city as much as the prototype CRSs that came out of the CGC would suffer in the analysis of truly real-world software. But these systems represented a start: the CGC revealed a glimpse of a possible future in which machines not only build our cars, drive us around, and manage our homes but also ensure the security and reliability of the software we use every day.

 Copublished by the IEEE Computer and Reliability Societies

We have discussed our CRS, Mechanical Phish, from a technical perspective in literature[1] and in a number of conference talks. In this article, we not only provide these technical details but also discuss the human side and organizational side of the creation of a CRS and the lessons that the CGC taught us about cyber autonomy.

The Cyber Grand Challenge

Traditionally, groups of humans faced off in capture-the-flag (CTF) competitions designed to push their hacking skills to the limit. In these competitions, each group is responsible for the defense of a networked computer. Because the computers defended by the teams have the same configuration and installed services, each team works on finding vulnerabilities in their instance, and then use the acquired knowledge to fix the found vulnerabilities—and, at the same time, break into the computers run by the other teams. Each successful hack produces a secret "flag," which is presented to the organizers of the competition to prove that the service has been compromised. Although it started as an event for pure enthusiasts, CTF competitions quickly evolved into something resembling more of an e-sport, with longstanding, well-known teams, corporate sponsorship, significant media coverage, and the occasional scandal or novel development to shake up the field.

The CGC was one such development. In the CGC, DARPA created a nearly traditional competition with one fundamental twist: no humans could take part. Instead, participants had to create a system that could reason about cybersecurity in a fully autonomous way. The idea was that these CRSs would face each other in a competition in which the human factor was completely removed, and only automated approaches that were able to deal with the complete identification-patch-exploitation pipeline could be used.

Feasibility Concerns

There are many challenges that must be surmounted when developing a CRS. Some of these—the prioritization of paths during symbolic execution, the improvement of precision during static analysis, and so on—require as-yet unknown scientific advancements to be solved. Others seem to be mostly engineering challenges, simply requiring a large development effort by many skilled developers.

One of the biggest engineering challenges facing CRSs is environment modeling. Certain binary analysis techniques (including symbolic execution, which was used by almost every CGC competitor) essentially perform an emulation of binary code on an exotic domain (that is, instead of reasoning about ones and zeroes as a normal CPU would, they deal with symbolic expressions, value ranges, and so on). These techniques need to be provided with models for the functionality of the environment, to represent the side effects of the actions performed by system calls. Unfortunately, modern operating systems utilize a wide range of such system calls (Linux has more than 300, for example), which makes the creation of these models tedious.

DARPA worked around this problem by creating the DECREE operating system, a simplified OS that contains just seven system calls:

- **terminate:** the equivalent of Linux's `exit()`
- **transmit:** the equivalent of Linux's `send()`
- **receive:** the equivalent of Linux's `recv()`
- **fdwait:** the equivalent of Linux's `select()`
- **allocate:** the equivalent of Linux's `mmap()`
- **deallocate:** the equivalent of Linux's `munmap()`
- **random:** the equivalent of Linux's `get_random()`

By simplifying the environment model, DARPA greatly lowered the barrier to entry, removing much tedious engineering effort from the development of CRSs. Otherwise, the environment was standard, using the well-studied and well-supported x86 architecture and a simple, custom binary file format (supporting only statically linked binaries).

CGC Qualifying Event

Because the CGC attracted more than 100 prospective teams, DARPA held a qualifying round, dubbed the CGC Qualifying Event, or CQE. One of these prospective teams was Shellphish.

Shellphish is a disorganized collection of hackers at the University of California at Santa Barbara computer security lab, and while the CGC was tangentially related to some of our research at the time, we could not devote much time to it. Thus, our CGC effort was more or less on the back burner until we could no longer ignore it—about two and a half weeks before the qualifying event.

In those two and a half weeks, we built a fledgling CRS, laying the groundwork for ideas that later turned into Driller[2] and Ramblr.[3] We built a vulnerability detection engine that combined the fuzzing techniques pioneered by American Fuzzy Lop (AFL)[4] with the symbolic execution capabilities of the `angr` framework.[5] In addition, we developed a patching engine that supported both "general" patches (when the CRS couldn't find a specific vulnerability to patch) and "targeted" patches (when it could).

The CQE differed from the final event in several ways. First, humans were allowed to monitor, start, and restart the CRSs but were not allowed to gain and use any knowledge from the binaries themselves. This made it less necessary to have a "bulletproof" system, because we could respond to system crashes. Second, actual

exploitation was unnecessary in the CQE—triggering a crash counted as "exploiting" a binary. This made it easier on the teams, in that they did not have to write an auto-exploitation component until after the CQE, but it also meant that the teams' patches had to prevent binaries from crashing, rather than simply making crashes unexploitable. Third, each CRS operated in isolation—there were no "flags" to capture from opponents, and scoring was purely on the basis of the crashing of the reference binaries in the dataset and protection against the reference exploits.

The CQE comprised a set of roughly 130 previously unseen binaries that the various CRSs had to analyze without any human involvement. Our CRS was able to crash 42 and prevent crashes in 49 of the CQE binaries. This, combined with the relatively high performance of the patches (which impacted the score), was enough to qualify us for the final event, netting us $750,000 in prize money.

CGC Final Event

The CGC Final Event (CFE) was very different from the CQE. The CRSs faced one another, needing to craft actual exploits (not just crashes), generate advanced patches with little overhead, steal flags, and adapt to the opponents' actions. There was more than a year gap between the CQE and the CFE to give teams enough time to develop their systems. True to form (and, again, because of the realities of a research lab), we procrastinated until the last three months.

The CFE was an incredible spectacle, in which the seven finalist CRSs (housed in seven massive racks provided by DARPA) competed live on stage, in front of an audience of thousands of people and with live commentary by "sportscasters." The humans of the teams watched it from the "team area," a cluster of couches within sight of the stage, but separated by a government-certified air gap.

There was absolutely no human intervention. The CRSs had to start on their own, hack on their own, and adapt to problems on their own. It was a grueling day, analogous in some small way to having to wait outside an operating room, with absolutely no control over what happens behind closed doors.

In the end, the Mechanical Phish won third place, netting us another $750,000 in prize money.

Birthing a CRS

What motivated us was the challenge of producing a fully integrated and robust system based on the current state of the art in binary analysis research. The difficulty of this challenge comes from the deep divide between "state of the art" and "robust," not just in technical terms, but in subtle cultural terms, too. Research labs, hanging on to the state of the art, are not normally well-known for the production of robust software. Instead, the incentive structure tends to favor the rapid creation and evaluation of "research prototypes," which work just enough to evaluate a given concept before moving on to the next research goal. Competing in something as consuming as the CGC is not a typical activity for a research lab. As such, we faced organizational and human challenges far beyond what we had been prepared for. While these challenges are not the type of technical details generally found in a scientific magazine, they are an important reality on our road to cyber autonomy.

We had to tackle designing an incredibly robust infrastructure, on hardware that we would not be able to access for issue remediation, at "move-fast-and-break-things" speed. We had to build a system that worked, without human intervention, for 10 hours.

From Research Prototype to Reliable Software

As academics, we are always chasing beyond the current cutting edge, to explore the next frontier. Because of this, the mode of operation in academic research is often to rapidly achieve the minimally functional prototype of an idea (without concern for beautiful design or reliability), evaluate it on a meaningful dataset, and publish the result. Normally, labs do not have (and do not need) the software development practices, ubiquitous in the industry, that encourage the development of *good* code. In fact, the term "research-quality code" has come to refer to code that showcases an idea but is almost unusable outside a research experiment. GitHub is rife with this kind of academically produced code, leading to much suffering among industry developers and enthusiasts who try to adopt it.

However, plenty of labs go against the grain, and our long history of creating services and software that work (such as Wepawet,[6] Anubis,[7] and `angr`[5]) is an attempt to provide to the public at large *usable* research prototypes. In the context of the CGC, the problem was that we did not have sufficiently good software development practices. This had to change on the fly—over the course of the CGC, we adopted practices such as continuous integration, issue tracking, and even an attempt at code freezes.

While this process was difficult, it had a humongous eventual payoff for our research. The direct benefit from the CGC was an extreme improvement in the reliability and performance of our binary analysis framework, `angr`. Since then, these improvements have been put to work powering a plethora of other research projects, both from our lab and from labs and companies around the world.

Human Organization

Through a process reminiscent of natural selection, our team settled into several main roles. We had a strategic leader, who oversaw the long-term direction and did the "people managing" (that is, the professor); the tactical captain, who managed the daily technical direction; and four technical teams to handle the infrastructure, the base binary analysis framework (`angr`), exploitation, and patching. These teams were logical, rather than physical entities—many of our teammates worked on more than one team throughout the CGC. For example, overlap between the base analysis framework team and the patching or exploitation team was fairly common.

Our team had a dozen people, none of whom had ever built a CRS before, and as mentioned earlier, we compressed the creation of the CRS into just over three months. Thus, the Mechanical Phish consumed just about three person-years of development. This is an area where the companies that were participating in the CGC had an advantage—from our understanding, the corporate teams had fewer members but were able to dedicate the entire two years of the challenge to their CRS development. In the end, as always, time was the most precious resource.

Making Use of Non-Temporal Resources

Other than time, for which we could have designed a better usage distribution, we also had to properly utilize a number of other resources. For example, DARPA provided a cluster of 64 extremely powerful machines for the development, testing, and eventual deployment of our CRS. Upon receipt of this hardware, we had a very vague idea of how the final version of Mechanical Phish would work. Thus, we designed an extremely flexible infrastructure, in which resources were automatically allocated as needed, using modern cluster management software (specifically, Kubernetes[8]).

This introduced a number of challenges. First, Kubernetes was (and remains) under extremely active development, so the base of our CRS was a moving target and needed periodic rewrites. Second, some of the CGC tooling that DARPA released (to work with binaries for the DECREE platform) required kernel modifications and ran only in a 32-bit VM (rather than a 64-bit container), necessitating the development of quite a bit of magic (actually consuming several hacker-weeks of development) to run virtual machines from within Kubernetes pods.

Complications

Naturally, complications arose throughout the process. Some of these were caused by our own disorganization or our attempts to surf the bleeding edge of security. Others were uncertainty issues that likely arise with any new competition format. Of course, the autonomy requirement of the final event, and the resulting inability to fix even minor issues arising from potential unexpected events, greatly amplified the stress caused by these complications.

Closed infrastructure. The most important part of building a system that can function autonomously is testing. As the various CRSs would talk to a central service (dubbed the Team Infrastructure; TI) during the game, the availability of this TI was necessary to test our systems. However, to avoid specific attacks developed against the TI, DARPA did not provide it to us in a runnable form. Instead, it provided a separate, partial implementation, called the Virtual Competition. The Virtual Competition implemented the minimal set of capabilities to start a game but did not have any functionality to evaluate exploits, test patches, generate sample network traffic, or compute scores.

Teams had to implement their own extensions to the Virtual Competition to have a readily available testbed, and DARPA did provide a network specification to help with this. As a result, there was no guarantee that these extensions were correct, or that they functioned in the same way as the actual TI.

Sparring partner uncertainty. The Virtual Competition was not enough to thoroughly test our systems. DARPA's solution to this was a set of "sparring partner" sessions, during which the actual TI would become accessible.

There was only one such interface, and eight entities clamoring to use it—the seven competitors and the infrastructure team. To prevent data leaks between these entities, the sparring partner could only be accessible to one of them at a time and had to be wiped between sessions. The result was that the sparring partner would, at an unannounced time, become accessible for an average of 30 minutes before shutting down. Unless the CRS was up, working, and properly scanning for the TI, the sparring session would be missed (this happened depressingly frequently).

Sometimes, this would cause interesting situations. One sparring partner round started in the middle of a database migration, with the central database of our CRS offline. To avoid wasting the sparring session, we launched off components of the CRS by hand, coordinating between them with a *paper database*, shown in Figure 1.

Thirty minutes was enough for five game rounds—sufficient to test basic CRS functionality but not CRS reliability. This meant that reliability issues, requiring a functional TI to trigger over a large amount of rounds, were very hard to identify. Worse, these five rounds had

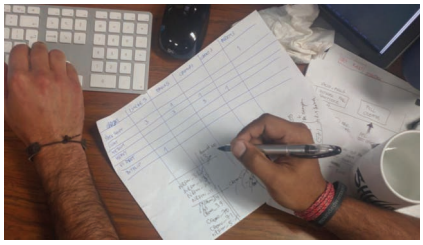


Figure 1. The paper database, used when a sparring partner session started in the middle of a database migration.

to be used to attempt to understand details of the performance scoring.

Performance scoring uncertainty. The CGC penalized teams for excessive performance overhead in their patched binaries. This performance score was critical—in fact, one of the teams that crashed the most binaries in the CQE failed to qualify precisely because their patches underperformed.

Naturally, determining the performance penalty was critical to the overall effectiveness of a CRS. However, two factors complicated this. First, DARPA did not specify exactly how performance overhead is calculated, and reproducing these calculations was very difficult. Furthermore, validating that the reproduced calculations were correct was impossible, as the relatively rare sparring partner sessions were the only way to get performance ground truth.

Second, DARPA kept the full penalty calculation formula (that took the time, memory, and file-size overhead and transformed it into a scaling factor applied to a team's points) secret. Without this formula, it was hard to reason about allowable performance tolerances for patching.

DARPA adopted this secrecy to stop teams from "gaming the system" ahead of time. This makes sense if there is a chance for human adaptability once the event begins. Without this chance, the secrecy made it extremely difficult for CRSs to make intelligent decisions about their patches.

Binary format uncertainty. One example of a small issue that caused great trepidation is the DECREE binary format itself. Into each CGC binary, the DECREE compiler tool-chain would insert a PDF file (which was always the same), along with a section of code that would checksum this PDF (by reading each byte of it when the program starts). This led to the obvious question: Will the PDF be in the binaries presented during the final event, as we remove the PDF to boost performance? Empirically, the answer to this question was "Yes," and the PDF seems to have been included to test whether the tools used were capable enough to properly remove it (and some binaries stressed the tools further by actually using data from the PDF outside of the checksum code) and reward such tools with lower performance overhead. However, DARPA stayed silent on the matter, requiring our tools to be adaptive to the no-PDF case, which did not manifest in the end.

Mechanical Phish

We have extensively described the various components of Mechanical Phish in research papers[2,3,5] and in an in-depth Phrack article.[1] For completion, we include a quick summary of the system high-level design. In Figure 2, one can see the overall layout of our architecture. The entire Mechanical Phish code base was composed by approximately 100,000 LoC (excluding external components), mostly written in Python. Out of these LoC, about 70,000 composed `angr`, the binary analysis framework on top of which most of the other components were built.

Infrastructure

DARPA provided every team 64 dedicated servers with overall:

- 1,280 physical cores,
- 16 TB of memory, and
- 64 TB of disk space.

To take full advantage of the hardware, we split our system into small, independent components and ran every component in a completely isolated environment. We used Docker containers to ensure components' isolation, ease of deployment, and scalability and Kubernetes to orchestrate the containers.

Every interaction with the game (for example, the retrieval of the current score or the submission of exploits) was performed through an API provided by DARPA, the TI. Ambassador and Network Dude were the components in charge of interacting with the game API and storing collected data into our central database, Farnsworth.

Meister and Scriba were the brain of our CRS. The first took care of reading the game status from Farnsworth and scheduling tasks. The second was responsible for deciding what patches and what exploits to submit, based on our internal evaluation and the feedback provided by the game API.

The hard program analysis work was done by the Workers, a set of components performing different tasks like bug finding, patching, exploitation, and results evaluation.

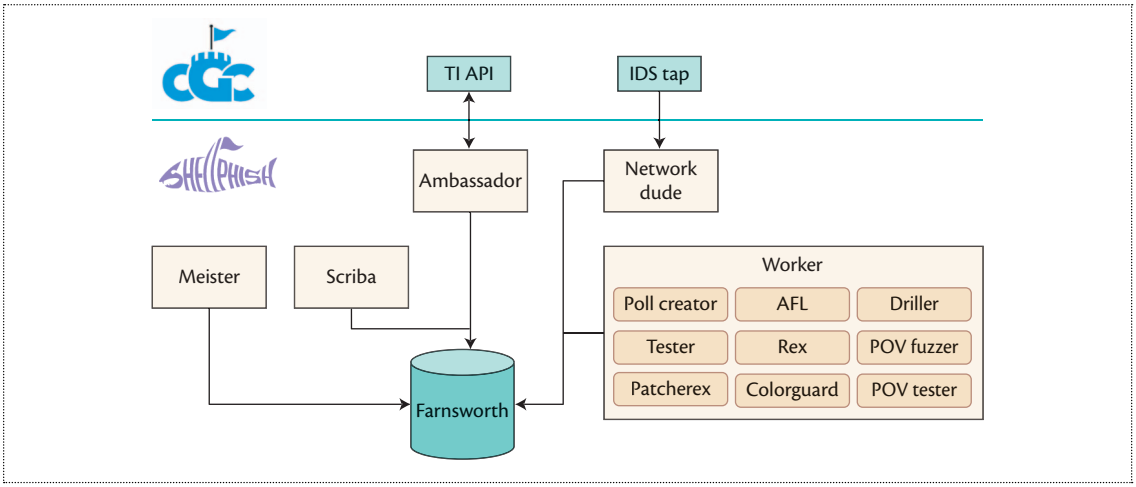


Figure 2. The architecture of Mechanical Phish.

Bug Finding

Mechanical Phish's exploitation involves two major steps. The first finds crashes in the target programs. The second step takes those crashes and attempts to figure out how they can be modified to produce exploits that take control of the program.

We used AFL, a well-known and highly successful evolutionary fuzzer, as the core of the bug-finding component of our CRS. For the CGC, we needed to handle a large variety of programs without any prior knowledge of what sort of inputs they will expect. An evolutionary fuzzer, such as AFL, is perfect for this, because it detects when inputs trigger new functionality inside the program, and then further mutates those inputs. This capability allows it to construct valid inputs, even when the program being fuzzed has strict requirements on the input format.

Although AFL is quite successful at finding bugs on its own, we found that it struggled to satisfy specific and difficult checks in the sample programs. Those checks could be as simple as matching a magic number or as difficult as solving an equation printed to the user. To handle these, we developed Driller, a tool that combines fuzzing with symbolic execution.[2] Symbolic execution is a slow but powerful technique for determining the equations that describe the state of the program at any point in execution. To use it efficiently, Driller limits the search space of the symbolic execution to that of the inputs generated by AFL. Specifically, the symbolic execution component will follow each input in AFL's corpus and check if there are any new locations in the program that it can reach.

AFL and symbolic execution, combined, made Driller highly successful in finding bugs that could be used to exploit the target programs.

Exploitation

The strategy we chose to exploit bugs found by the Driller component was to first analyze the crash using symbolic execution. That is, we symbolically traced the program following the crashing input, and when we got to the crash, we modified the input as needed to make an exploit.

In general, it can be extremely complicated to figure out how to exploit a particular bug or crash. For Mechanical Phish, instead of trying to design a general strategy, we came up with a list of crash types that we could exploit and methods to exploit just those particular crashes. The crashes we targeted were instruction pointer overwrite, arbitrary read address, arbitrary write address, and vtable overwrite. That list of crashes was picked specifically to target a fairly large range of what we expected to see in the CGC. Many types of bugs could map to the same crash type. For example, IP overwrite could occur from a buffer overflow, a use after free, an out-of-bounds index, and so forth. In addition, for each type of crash, there were multiple techniques that would try to exploit it in different ways.

After tracing the crash, Mechanical Phish would apply each technique and check if it succeeded in making a working exploit. Eventually, when one was found, Mechanical Phish would begin using it against the opponents.

Here we show a function with a basic stack overflow.

```
void say_hello() {
 char name[20];
 read_string(name);
 printf("hello %s\n", name);
 return;
}
```

In a normal interaction, the name provided will be short enough to fit entirely in the name buffer and

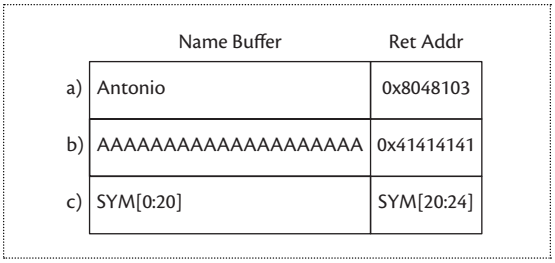


Figure 3. The function say_hello() from the listing in the main text has a buffer overflow. This figure shows the stack of the buffer during the following: (a) normal interaction, (b) overflowing input, and (c) the symbolically traced input.

the program will execute as expected. For that case, the stack of the function will look like the example in Figure 3a. However, the fuzzing component in Mechanical Phish can easily generate an input that is too long and overflows the return address, causing the program to crash (Figure 3b). Next, Mechanical Phish will symbolically trace the crashing input as shown in Figure 3c. It will understand that at the crash the instruction pointer is equal to `SYM[20:24]`, where `SYM` is used to denote symbolic input.

To exploit this, Mechanical Phish would try to jump to the bytes we control and execute them as code, referred to as *shellcode*. It added constraints to the equations that were collected during symbolic tracing. First, it placed the shellcode in memory by adding the constraint `SYM[0:20] == shellcode`. Then it constrained the overflowed return address to be the address of the shellcode, `SYM[20:24] == addr(shellcode)`. Finally, it asked the constraint solver to generate an input that matches these equations; this input was our exploit.

Patching

Patcherex, which is built on top of `angr`, is the central patching system of Mechanical Phish.

Patcherex follows an untargeted approach. In other words, it modifies binaries by applying generic binary hardening techniques, without using directly any knowledge about how a binary is exploitable. Nevertheless, in many cases, these hardening techniques are able to make vulnerabilities initially present not exploitable.

Furthermore, even when these vulnerabilities are still exploitable, the way in which exploits have to be carried out changes significantly in many patched binaries. For this reason, in many cases opponents were forced to analyze our patched binary to be able to adapt their exploits. However, we also implemented binary modifications hindering static and dynamic analysis of our patched binaries, making automatic analysis extremely hard, if not impossible. These included both passive countermeasures (that is, the produced binary files were slightly corrupted, being able to be executed in the DECREE environment but not analyzed with gdb or IDA) and active countermeasures. For example, we identified a buggy instruction in the floating-point support of the QEMU emulator that, when specific conditions were met, would cause the process to freeze. The inclusion of this instruction in our patches would hang any systems based on QEMU (in fact, the visualization system used by the organizers in the final event actually froze due to this countermeasure when visualizing an attempted exploit against one of our patched binaries).

Given the scoring system of the CGC competition, the primary concern while developing Patcherex was not to degrade the functionality of the original binaries and their performance. In fact, while it is reasonably easy, in general, to harden a binary to make it not exploitable, it is extremely hard to achieve this goal without significantly affecting its performance. Furthermore, compiled code often presents corner cases (due to, for instance, compiler optimizations) that, if not handled correctly during patching, will lead to the generation of nonfunctioning code.

Patcherex applies to any analyzed binary a list of techniques that corresponds to high-level patching strategies. Applying a specific technique to a binary generates a set of patches, which are low-level descriptions of how a fix or an improvement should be made on the target binary, such as adding some code or data to the original binary.

We implemented three different types of techniques:

- *Binary hardening*: Generic binary hardening techniques. For instance, we implemented encryption of the return pointer and a "loose" form of control flow integrity. We also implemented a technique to prevent memory-leaking exploits. In particular, we added code to the patched binary to check the transmitted data.
- *Anti-analysis*: Techniques aiming to prevent rivals from analyzing or stealing our patched binaries. For instance, we specifically added code triggering QEMU emulation bugs. In addition, we also inserted a back door in our patched binaries so that, in case they were reused by any opponent team, we could have trivially exploited them.
- *Binary optimization*: We realized that many of the provided binaries were easily optimizable (mainly because they were originally compiled without using compiler optimizations). Therefore, we applied binary optimization techniques (such as constant propagation or dead assignment elimination) to lower their memory/CPU usage. Improving the performance of the original binaries allowed us to lower the

negative impact (in terms of performance and, as a consequence, score) that the addition of patches generated by the previously mentioned techniques inevitably introduced.

Patches generated by applying the different techniques to a binary were then integrated into the original binary by a patching backend. Specifically, we developed a "reassembler" backend, able to convert a binary from its compiled form to an assembly form (recovering, for instance, function boundaries, function pointers, and pointers to data structures in memory). This form allows us to easily add or modify existing code and data and then use existing assemblers to generate a patched binary. Full details about the reassembler backend have been published in an academic paper.[3] As a fallback solution, in case the reassembler backed fails to generate a working patched binary, we use a different backend. This alternative approach is based on the inline insertion of detours (that is, *jmp* instructions), and it generates patched binaries that are less likely to misbehave, but slower and more memory greedy.

> **The drive to win against other teams is a stronger motivating force than a research deadline or the need to achieve some abstract result.**

Lessons Learned

Participating in the CGC taught us a number of lessons, both technical and nontechnical, which shaped our research and the pursuit of similar endeavors.

Teamwork

Effective teamwork is essential. A graduate student lab might not have the discipline of the well-managed development group of a company, but it has a unique drive and a camaraderie that cannot be easily replicated. Even though we suffered some setbacks due to the lack of experience in the development of high-quality software, the team was able to step up to the task without concerns about personal-life side effects. This is what a competition, like the CGC and many human-based CTFs, fosters: the drive to win against other teams is a stronger motivating force than a research deadline or the need to achieve some abstract result. On the other hand, these engagements cannot be the norm, as the toll (in terms of stress and pure physical exhaustion) that these kinds of events bring is not sustainable.

Gaming the Game

Understanding the nature and rules of the game is essential. Interestingly, the top-scoring system, Mayhem, had a dramatic failure in the middle of the competition, which prevented the system from finding new exploits against other teams. However, by not doing anything and simply passively defending, the Mayhem system was able to maintain its advantage against the other CRSs, winning the competition.

On our side, we were undecided between two different approaches to pushing patches. This was an important part of the game, as pushing a new version of a binary came with a one-round penalty in terms of defense points. As a result, pushing binaries too often could result in a substantial loss of points.

Our two possible approaches were the following:

- *Always-patch strategy*: Push patched binaries as soon as we were sure that their performance was acceptable.
- *Patch-if-exploited strategy*: Push patched binaries as soon as we were sure that they were performant enough and we had developed an exploit for the vulnerability.

The second approach was motivated by the fact that we assumed that most teams would have the same (or at least a very close) capability for exploitation. Under this assumption, the fact that we found an exploit for a specific target binary would imply that it was highly likely that other teams would have found an exploit as well, and therefore, it was reasonable to push a patched binary and take the associated defense penalty.

A few hours from the beginning of the competition, somewhat emboldened by the fact that our patching seemed to be highly effective with minimum performance overhead, we decided to push patched binaries as soon as we were able to produce them (that is, we chose to use the always-patch strategy). This decision resulted in a penalty that cost us the victory, as our postgame analysis revealed.

In this regard, it is very important to point out that every team could look back and consider things that they might have done differently. Understanding what the best strategy "would have been" is easy after the game is over. On the contrary, before the game, many aspects were unknown (for instance, how many challenges will be exploited), and therefore, choosing an optimal strategy was significantly harder.

Our postgame analysis was performed by computing scores for several simulated CGC rounds where the

Mechanical Phish undertook different strategies. The results are as follows:

- *Patch-if-exploited strategy*: We calculated our score with a patch strategy that would delay patches until after we launched exploits on the corresponding binary. In this case, our score would have been 271,506, putting us in first place.
- *Never-patch strategy*: We assumed that any time an exploit would be launched on a binary against any team, the exploit would be run against us during that round and all subsequent rounds. With this calculation, our score would have been 267,065, putting us in second place.
- *No-op strategy*: We ran an analysis similar to the never-patch strategy, but we also removed any exploitation-provided points. In this case, we would have scored 255,678 points, barely beating Shellphish and placing third in the CGC.

Exploitation

We were surprised that our exploitation system turned out to be the most effective of any competitor's during the final event, in terms of both the unique number of exploits produced and the number of times an exploit successfully worked. We exploited 15 different challenges, while the next best competitor exploited 11. However, there were 82 total programs, so what kept us from exploiting more? First, there were a good number of errors in our implementation. But, other than those, we believe that automatically exploiting bugs requires more than the "bag of techniques" approach we developed.

In exploitation, it is common that a human will carefully set up the program state, such that when the bug is triggered, structures and memory are already correctly set up. Our approaches did not have any way to backtrack and trigger the other functionality before that bug that would aid in setting up the state correctly. For some cases, this implies that we had to hope that the fuzzing component generated a crash where the state was already set up correctly, and this was not always the case.

Binary Patching

Many techniques exist for binary patching, including in-place bytes replacement, detouring, and so on, as well as systematic patching solutions like static binary rewriting techniques and dynamic binary instrumentation.[9–11] However, the CGC setting imposed some vital restrictions: The customized OS (DECREE), which has a very restricted set of system calls and a significant lack of system mechanisms (like process forking and debugging), made any dynamic approach unusable. Moreover, the tight overhead allowances for both performance and file size prevented us from applying many static binary rewriting techniques, which either unacceptably degrade the overall performance of patched binaries or add a noticeable amount of extra bytes to provide safety guarantees for rewriting. Hence, we picked *reassembling* (or *reassembleable disassembling*[12]) as the major binary rewriting technique and implemented Ramblr, with detouring as a fallback.

Some facts about the binaries in the CQE make reassembling a natural choice: All binaries are self-contained—no library is needed at all. Nearly all the binaries are compiled without any optimization flags switched on. Most of the binaries are relatively small compared to real-world targets, like word processors or browsers. Last but not least, only a few binaries are obfuscated, and it is not difficult to identify this problem and bail out. These facts made our CFG recovery and code data differentiation—which are the foundation of many static analyses, including reassembling—much easier.

After the CFE, we successfully applied Ramblr on more targets—including many CTF binaries—for binary rewriting and patching. Nevertheless, it is worth noting that all target binaries we rewrote using Ramblr were not considered "huge." We believe that using Ramblr on large or complex binaries will not yield a satisfactory result, as code data differentiation becomes harder when the code base gets larger, and our reassembling approach is best-effort and empirical—it does not provide safety guarantees (that is, it does not guarantee that no immediate value is treated as a pointer during reassembling). As we see it, providing safety guarantees is very difficult, if not entirely impossible. Therefore, Ramblr, in its current form, does not seem to be an ideal choice for rewriting large, real-world targets.

Infrastructure

Our bug-finding techniques pushed the limits of the bleeding-edge DARPA-provided servers. During our tests, processes died because the system ran out of memory regularly, and entire servers became unresponsive because of CPU-intensive workloads.

Normally, human intervention can mitigate these problems quite easily. However, during the CGC Final Event, our CRS had to run completely autonomously, which is why we invested a substantial amount of time in creating a highly available and fault-tolerant system.

Containers, which we orchestrated through Kubernetes, were the core foundation of the Mechanical Phish. To facilitate proper recovery without losing too much data, we designed our components to be stateless, and we broke down the complex functionality of our CRS into smaller components that executed separately, and whose results could be check-pointed and stored. Thanks to this design and by leveraging the tools that

Kubernetes provides, server failures were not critical. In fact, in our tests, Mechanical Phish kept exploiting and patching even if up to two-thirds of the cluster failed.

Unfortunately, stateless services are only one side of the coin, and the most important components are not stateless, namely the Kubernetes API server itself and our database. For these only two stateful components, we deployed multiple redundant instances with fail-over running on different nodes.

Interestingly, when we started to design the architecture of our CRS (in December 2015), Kubernetes was still in an early stage (version 1.0, July 2015). Since then, and most notably during our development process, Kubernetes has seen significant development and many improvements have been made. Although a blessing, this was a curse at the same time: constant API changes and updates broke compatibility, and our code base had to be dealt with on a regular basis.

Regardless of our problems during the development for Mechanical Phish, Kubernetes was easy to use and powerful. In fact, after our positive experience, we converted our research lab from a system where users have bare-metal servers allocated to them, to a container-based system where users request CPU and memory on an ephemeral basis, improving our overall resources utilization significantly and allowing research experiments of significantly larger scale than ever before.

The autonomous system determines which analysis actions need to be carried out by its components. Then, the system creates tasklets, some of which can be delegated to humans.

Aftermath: DEF CON CTF

When the DARPA CGC was announced, LegitBS,[13] who were the organizers of the 2016 DEF CON CTF, decided to structure the competition in a way that was identical to the CGC, so that the CRS that would win the CGC could compete against human teams. As a result, the Mayhem CRS was one of the teams playing in the 2016 DEF CON CTF.

However, Shellphish was the only team that qualified for both the CGC and the DEF CON CTF, and therefore, we had a unique opportunity: we could have Mechanical Phish play alongside humans.

Mechanical Phish was able to observe how humans (that is, the Shellphish team members) interacted with a target application when they were trying to find vulnerabilities. Then, the system used these interactions as seeds for its own vulnerability analysis process, with surprising results. On many occasions, the system was able to leverage the human inputs to reach "deep" into the application and identify vulnerabilities that could not have been identified without human assistance. Interestingly, the CRS did more than simply play backup to its human partners. Rather, it used human input to enhance its own ability and beat the humans to the punch: more than half the vulnerabilities found by the combined team were created by Mechanical Phish after leveraging human input to guide its analysis.

The successful interaction between the automated reasoning system and the human analysts prompted a key observation. Throughout the history of the field of vulnerability analysis, the principal paradigm has been the use of *tool-assisted human analysis*, in which human analysts would carry out the core analysis tasks, while utilizing automated techniques as an aid. In this case, the humans are the orchestrators of the analysis process, and they delegate specific tasks to specific tools, taking care of combining and composing the results of multiple tools. The CGC pushed a second approach: complete automation, where fully automated strategy routines utilized fully automated analyses to identify, exploit, and patch flaws in software. This inspired a third, heretofore unexplored model, which is *human-assisted automated analysis* of software. In this model, in an inverse of current techniques where most approaches see automated tools as an aid or extension to human analysts, human analysts can instead be used as an aid to automated vulnerability analysis systems.

Following this approach, the autonomous system determines which analysis actions need to be carried out by its components. Then, the system creates *tasklets*, some of which can be delegated to humans with different skill levels (for instance, experts or nonexperts). Even though this approach is still in its infancy, our preliminary results show that by orchestrating humans in a large-scale complex vulnerability analysis process, it is possible to identify vulnerabilities that would not be identified by purely automated means, shining a new light on one of the most challenging problems in program analysis.[14] ■

References

1. "Cyber Grand Shellphish," Shellphish, Jan. 2017; http://shellphish.net/cgc.
2. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," *Proceedings of the Network and Distributed System Security Symposium* (NDSS 16), 2016.

3. R. Wang et al., "Ramblr: Making Reassembly Great Again," *Proceedings of the Network and Distributed System Security Symposium* (NDSS 17), 2017.
4. M. Zalewski, "American Fuzzy Lop," 2017; http://lcamtuf.coredump.cx/afl.
5. Y. Shoshitaishvili et al., "(State of) The Art of War: Offensive Techniques in Binary Analysis," *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.
6. M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code," *Proceedings of the 19th International Conference on World Wide Web* (WWW 10), 2010.
7. U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A Tool for Analyzing Malware*, EICAR, 2006; https://www.cs.ucsb.edu/~chris/research/doc/eicar06_ttanalyze.pdf.
8. "Kubernetes," Google, 2014; https://kubernetes.io.
9. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 05), 2005, vol. 40, p. 190.
10. N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 07), 2007, p. 89.
11. M. Smithson et al., "Static Binary Rewriting without Supplemental Information: Overcoming the Tradeoff between Coverage and Correctness," *Proceedings 20th Working Conference on Reverse Engineering* (WCRE 13), 2013, pp. 52–61.
12. S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," *24th Usenix Security Symposium* (USENIX Security 15), 2015, pp. 627–642.
13. L.B. Syndicate, "Legitimate Business Syndicate CGC Documentation," 2015; https://cgc-docs.legitbs.net.
14. Y. Shoshitaishvili et al., "Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance," *Proceedings of the ACM Conference on Computer and Communication Security* (CCS 17), 2017.

Yan Shoshitaishvili is an assistant professor at Arizona State University, where he leads research into automated program analysis and vulnerability identification techniques. Contact at zardus@shellphish.net.

Antonio Bianchi is a PhD candidate at the University of California at Santa Barbara. Contact at antoniob@cs.ucsb.edu.

Kevin Borgolte is a PhD candidate in the Computer Science department at the University of California at Santa Barbara. Contact at cao@shellphish.net.

Amat Cama is a world-renowned hacker, having participated in countless CTFs around the globe. Contact at amatcama@gmail.com.

Jacopo Corbetta is a senior engineer at Qualcomm Product Security. At the time of this writing, he was an independent researcher. Contact at jacopo.corbetta@gmail.com.

Francesco Disperati is a senior software engineer at PayJunction. Contact at me@nebirhos.com

Audrey Dutcher is an undergraduate computer science researcher at the University of California at Santa Barbara. Contact at dutcher@cs.ucsb.edu.

John Grosen is a computer science major at the Massachusetts Institute of Technology. Contact at jmg@johngrosen.com.

Paul Grosen is a high school student, researcher in the Security group in the Department of Computer Science at the University of California at Santa Barbara, and a Shellphish member. Contact at pcgrosen@cs.ucsb.edu.

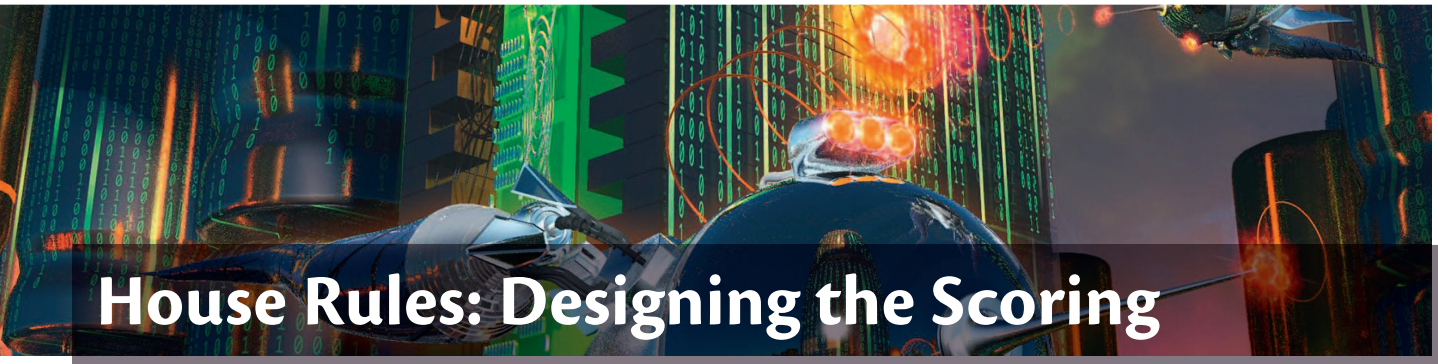
Aravind Machiry is a PhD candidate at the University of California at Santa Barbara. Contact at machiry@cs.ucsb.edu.

Chris Salls is a PhD student at the University of California at Santa Barbara, where he works on automated techniques to find memory corruption bugs. Contact at salls@cs.ucsb.edu.

Nick Stephens is a security researcher and a member of the Shellphish team. Contact at nick.d.stephens@gmail.com.

Ruoyu "Fish" Wang is a PhD candidate at University of California at Santa Barbara. Contact at fish@shellphish.net.

Giovanni Vigna is a professor in the Department of Computer Science at the University of California at Santa Barbara, the CTO at Lastline, Inc., and the founder of Shellphish. Contact at zanardi@shellphish.net.



House Rules: Designing the Scoring Algorithm for Cyber Grand Challenge

Benjamin Price and Michael Zhivich | MIT Lincoln Laboratory
Michael Thompson and Chris Eagle | Naval Postgraduate School

The key driving force behind any capture-the-flag competition is the scoring algorithm; the Cyber Grand Challenge (CGC) was no different. In this article, we describe design considerations for the CGC events, how these algorithms intended to incentivize competitors, and effects these decisions had on the resulting gameplay.

Scoring algorithms are at the core of any competition. They define the objective of a game and drive competitors' strategies, guiding investments of effort, affecting resulting gameplay, and last (but not least) determining who captures the glory and walks away with the prize. As such, much consideration is given to design of scoring algorithms by organizers of any competition, and the Cyber Grand Challenge (CGC) was no different. CGC's lofty vision was to "engender a new generation of autonomous cyber defense capabilities that combine the speed and scale of automation with reasoning abilities exceeding those of human experts."[1] In particular, CGC challenged competitors with a highly nontrivial task of "improv[ing] and combin[ing] semi-automated technologies into an unmanned Cyber Reasoning System (CRS) that can autonomously reason about novel program flaws, prove the existence of flaws in networked applications, and formulate effective defenses."[1]

In practice, this meant that a CRS would be provided a bespoke, known vulnerable, *challenge binary* that implemented functionality for a never-before-seen network application or service. The CRS would then have to produce a *replacement binary* that mitigated the effects of the embedded vulnerability and a *proof of vulnerability* describing an interaction with the vulnerable binary that would cause it to exhibit undesired behavior (for example, crash, leak sensitive data, and so on).

The infrastructure team organizing this competition arguably had an equally daunting task of creating a sufficiently realistic, yet distinct and protected arena in which these CRSs would compete in the first fully automated attack–defense capture-the-flag (CTF) event. When developing the scoring algorithms that would guide development of the first crop of automated cyber reasoning systems, we focused on ensuring the following properties of scoring:

- *Fairness*. Scoring should not discriminate against a specific method the team uses to solve the problem. Note that artifacts that appear in the solution could be penalized (for example, using too much memory in a replacement binary), but no specific process of bug discovery and remediation should be prescribed or proscribed.
- *Collusion resistance*. We wanted to entice competitors to focus on improvements in automated network defense, not analysis and defeat of the scoring

1540-7993/18/$33.00 © 2018 IEEE Copublished by the IEEE Computer and Reliability Societies

algorithm. In particular, the scoring algorithm should disincentivize collusion between participants.
- *Real-world relevance*. We wanted the results of this research to produce practical prototypes that could be readily adopted by the industry, so the scoring algorithm aimed to replicate many of the pressures the real world places on security solutions.
- *Automated evaluation*. Because CGC is a machine-scale competition, it would be impossible to score the results by hand if only due to the number of submissions. Thus, scoring cannot rely on "expert judgment" of any kind; all measures required for scoring have to be automated.

Armed with these principles, we considered many different options, and settled on two variants of the same general algorithm—one used for the CGC Qualifying Event (CQE), and one used for the CGC Final Event (CFE); the variant algorithms reflect differences in the number of participants and mechanics of the competition between CQE and CFE.

In the rest of the article, we describe the algorithms themselves, the reasons these formulations were selected, how the algorithms affected gameplay in CQE and CFE, and general lessons learned that might be drawn from our experience to help other CTF competition designers.

Scoring Rubrics: What to Measure?

The first question that comes to mind when creating the scoring algorithm is: What should we measure? Given that the CGC competition is about improving the state of the art in automated network defense, we certainly need some measure of security provided by the replacement binaries produced by competitors' CRSs. However, it is exceedingly easy to provide a binary that has perfect security—one that does nothing. Therefore, we also need to measure the availability of service provided by the replacement binary. Finally, because CGC is a competition focused on creating cyber reasoning systems, the scoring algorithm should reward CRSs that can find a vulnerability in the original binary—we term this rubric *evaluation*.

Security

The world would be a saner place if an analytic technique existed that could examine an application and provide a complete listing of embedded exploitable vulnerabilities; in such a world, we would not need a Cyber Grand Challenge, and we would not suffer from so many cyberattacks. Instead, we have to consider security as a relative, not absolute metric. Attacks provide a concrete demonstration of the vulnerability on which to base measurement.

In CGC, two different entities provided proofs of vulnerability (PoVs), which represented attacks in our game environment: challenge binary (CB) authors and competitors' cyber reasoning systems. Challenge binary authors were required to provide a PoV that demonstrated the exploitability of each vulnerability embedded in a CGC challenge binary. The CRSs of course would also discover and prove vulnerabilities in the challenge binaries that formed the substrate of the competition.

Due to the different provenance of PoVs, two different security scores were created: a *reference security* score that measured how well a replacement binary defended against PoVs provided by CB authors, and a *consensus security* score that measured how well a replacement binary defended against PoVs provided by CRSs.

Availability

As mentioned previously, providing a "perfectly secure" replacement binary would be trivial, if the replacement were not tested for functionality. Clearly, this would violate the guideline to encourage solutions that have real-world applicability, so we needed a way to measure the functionality of the replacement service. To accomplish this task, we relied on CB authors to provide a test case generator that could automatically create thousands of test cases for the application they have developed. Each test case included not only the input to the application under test but also the logic to decide whether the application's response was correct. In effect, the test suite created an automatically checkable specification of the application behavior. A measure of functionality was determined by the number of tests passed by the replacement binary compared to the number of tests passed by the original application.

However, measuring only functionality does not fully satisfy the real-world applicability requirement. In practice, *performance* of an application or service is also of great importance, and many security solutions have not found wide adoption due to their performance overhead. To measure performance impact, we concentrated on three typical performance factors: CPU execution time, memory usage, and file size. The former two metrics were computed in aggregate over the execution of the test cases used as part of the functionality test, whereas the latter involved a simple comparison of file sizes between original and replacement binaries.

Evaluation

Finally, because we wanted to reward creation of cyber reasoning systems, we awarded additional points for finding vulnerabilities. The most indisputable way to prove that a vulnerability exists is to provide an input or interaction with an application that causes the application to exhibit some bad behavior (for instance, crashing or revealing sensitive information). In CGC, this input

took a form of an XML-based description (for CQE) or a full binary that interacted with an application over a network socket to prove vulnerability (in CFE).

Scoring Algorithm

The scoring algorithm for CQE and CFE shared the same structure, though different considerations were prioritized when defining each term. The overall algorithm scored each challenge set individually and summed the scores over challenge sets (for CQE) and over both challenge sets and rounds (for CFE). A challenge set score was defined as a product of the scoring rubrics described, because we wanted to signal that all parts of the competition are equally important and excelling at just one aspect would not be sufficient to win:

$$Score(RB, PoV) = Avail(RB) \times Sec(RB) \times Eval(PoV).$$

The *availability* term was defined similarly between CQE and CFE; however, the *security* and *evaluation* terms differed slightly. To explain the different formulations for these score components, let's discuss each event separately.

CGC Qualifying Event

CQE was held halfway through the program as a way to downselect the cyber reasoning systems that would participate in the final, completely automated, attack–defense capture-the-flag competition. Because the event occurred relatively early in the program, it was not conducted as a full-scale attack–defense competition, but instead as an approximation of a single round in the final event. During this event, the cyber reasoning systems were provided a set of 131 previously unseen binaries with embedded vulnerabilities. Some binaries came with a small capture of network traffic that could be used to deduce their intended functionality. The CRS had 24 hours to analyze the binaries, produce replacements that would mitigate embedded vulnerabilities, and provide PoVs (in CQE, this was an input that would cause the vulnerable binary to crash with a segmentation violation or illegal instruction fault). Each entry was then scored according to the above algorithm (summed over all entries to produce the final CQE score), where the terms were defined as described below.

Availability.

$$Avail(RB) = min(Perf(RB), Func(RB))$$

$$Perf(RB) = max\begin{cases} mem\_use\_overhead(RB) \\ cpu\_exec\_time\_overhead(RB) \\ 0.25 \times file\_size\_overhead(RB) \end{cases}$$

$Func(RB) =$ **(Number of successful tests)/(Number of tests)**

As discussed previously, availability measures the performance and functionality of the replacement binary and assigns an availability penalty based on whichever degradation is worse. To combine effects of file size, memory usage, and CPU execution time overhead, a similar approach is taken—the maximum of the three performance overheads is selected. In addition, both *Perf* and *Func* scores are passed through functions that provide a "faster-than-linear" decrease in score (see Figure 1a and Figure 1b, respectively). The idea behind these curves was to provide better differentiation between higher-performing teams and ensuring that a smooth transition is made from maximum availability score of 1 to minimum of 0. Note that the *Perf* graph provides a 10 percent "grace factor"—that is, if the performance of the replacement binary is within 10 percent of the performance of original binary, then no performance penalty is applied. At the other end, a performance overhead of 100 percent or more results in 0 availability score.

Security.

$$Sec(RB) = \begin{cases} if\ SecRef(RB) > 0: \\ 1 + \frac{1}{2} \times (SecRef(RB) + SecCon(RB)) \\ otherwise: 0 \end{cases}$$

In CQE, security was evaluated both against the PoVs provided by CB authors (*SecRef* for reference) and CRS-provided PoVs (*SecCon* for consensus). The former provided an unbiased measure of whether the CRS has mitigated exploitability of a bug inserted by a challenge binary author; at least one PoV in this category had to be mitigated by the replacement binary in order for security score to be non-zero. Because other PoVs for the same bug might exist, the competitor-provided PoVs were also tested against all replacement binaries for that challenge set. If a replacement binary mitigated all competitor PoVs, its *SecCon* score was set to 1; otherwise, it was 0.

The reasons for this formulation of the security score derive from the requirement that the scoring algorithm be collusion resistant. Because CQE was conducted online and open to a large number of participants, we were concerned that a team might register many "sock puppet" teams that would submit PoVs that the "master" team would know how to defend, thus artificially inflating its security score while providing no additional security. Therefore, we could not give equal weighting to reference PoVs and consensus PoVs. A detailed description of the red-teaming exercise that led to this decision can be found in the CGC FAQ.[2]

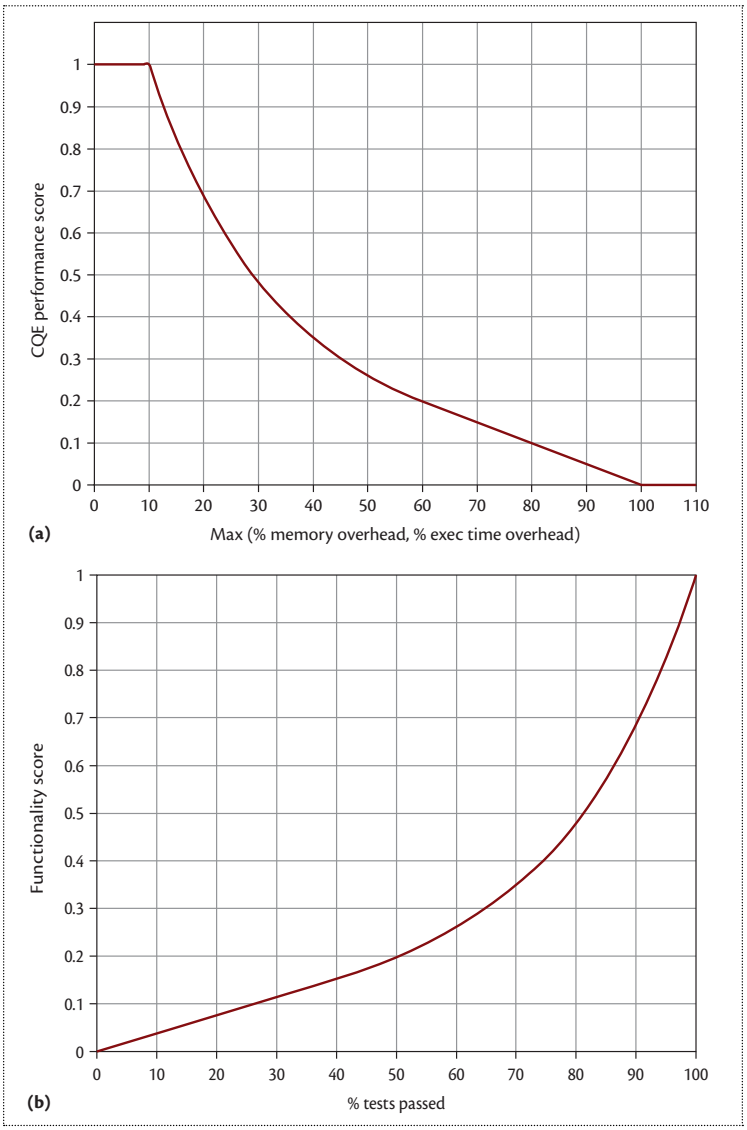


Figure 1. Curves illustrating conversions for (a) performance and (b) functionality.

Evaluation.

$$Eval(PoV) = \begin{cases} 2 & Submitted\_PoV\_successful \\ 1 & otherwise \end{cases}$$

The evaluation portion of the score was straightforward: teams were awarded a 2x multiplier for providing a successful PoV against a reference challenge binary, thus increasing that team's score. Note that providing a working PoV may also decrease a competitor's score by removing their SecCon bonus.

Discussion. Putting all the terms together, we have the following possible score values:

- *Balanced CRS*. A CRS that solves the challenge completely with perfect retention of functionality and performance (and a successful PoV) would receive a score of $1 \times 2 \times 2 = 4$.
- *Defense-only CRS*. A CRS that uses a defense-only strategy with perfect retention of functionality and performance would receive a score of $1 \times 2 \times 1 = 2$.
- *Offense-only CRS*. A CRS that uses an offense-only strategy would receive a score of $1 \times 0 \times 2 = 0$.
- *Do-nothing CRS*. A CRS that just returned the original vulnerable binary would receive a score of $1 \times 0 \times 1 = 0$.

The general idea behind the CQE scoring algorithm is that the teams that could automatically mitigate PoVs while maintaining functionality and performance of an application should receive a high score. If they can also provide a PoV against the original binary, their score is increased and their competitors' consensus scores might decrease. We wanted to reward finding vulnerabilities, but not preclude defense-only solutions. Hence, the evaluation factor will not cause the score to be 0; however, a purely offense-oriented solution was deemed insufficient, so a team could not score points by just providing a PoV, without associated nontrivial defense in the replacement binary. Thus, we were selecting teams with both good defense and good offense to advance to the finals.

CGC Final Event

CGC Final Event was a completely autonomous attack–defense CTF among the seven finalist CRSs determined by total CQE scores. The structure of CFE differed significantly from CQE—it consisted of 96 rounds during which new challenge binaries could be introduced into the game or old ones retired. Each challenge binary fielded by a CRS was evaluated for functionality and performance, with feedback provided to the CRS. Each CRS had an opportunity to download replacement binary and network filter submissions made by other competitors during the previous round and evaluate them to find any new or remaining vulnerabilities. Each CRS could also submit a PoV to be launched against a specific competitor and receive feedback of its success or failure.

There were two different kinds of PoVs in CFE: Type 1 (controlled crash) and Type 2 (memory disclosure), which modeled different attack types. Type 1 PoV required that the vulnerable binary would crash with specific values of instruction pointer and a general-purpose register that were prenegotiated with the game infrastructure—this was a proxy for *remote code execution* attack. Type 2 PoV required that an interaction with the vulnerable binary result in leaking 4 consecutive bytes from the so-called magic page that was mapped at a fixed address and contained

specially crafted pseudorandom data that changed each time the binary started—this was a proxy for a *secret stealing* attack. A more detailed description of CFE PoVs can be found in "Proof of Vulnerability Markup Language."[3]

The overall structure of the scoring algorithm remained the same as in CQE (the scores were now summed over both challenge sets and rounds to arrive at the final score); however, several of the terms required adjustments to capture the new complications of rounds, consensus evaluation of security, and introduction of network-based defenses.

Availability.

$$Perf(RB) = \max \begin{cases} mem\_use\_overhead(RB) \\ cpu\_exec\_time\_overhead(RB) \\ wall\_time\_overhead(RB) \\ 0.25 \times file\_size\_overhead(RB) \end{cases}$$

The performance portion of the availability term was slightly adjusted to include a new component of wall-clock time in addition to existing CPU execution time. This new component reflected the measurement of slowdown incurred by using a network-based filter to detect or respond to attacks; the wall time was measured at the host performing the functionality test. In addition, the grace factor was lowered from 10 percent to 5 percent to correspond more directly to existing industry requirements for readily adoptable security solutions.[4]

Furthermore, to enable consensus evaluation of replacement binaries, CFE infrastructure imposed a one-round delay on running the replacement binary or associated network filters. This meant that a team replacing a binary would get an availability score of 0 for the service being replaced in that round, while the other competitors would get an opportunity to analyze the binary for new or remaining vulnerabilities. This dynamic was intended to emulate real-world practices where downtime is very costly and rarely tolerated.

Security.

$$Sec(PoV) = \begin{cases} 1 & any\_competitor\_PoV\_successful \\ 2 & otherwise \end{cases}$$

The security term changed as well. Because we expected the replacement binaries to change in response to attacks from other competitors, it no longer made sense to evaluate security using reference PoVs provided by CB authors as they were unlikely to work against the replacement binaries after the first round. Therefore, this score component reflected a more empirical notion of security: if any competitor successfully proved vulnerability in the replacement binary, the score was set to 1; otherwise, it was set to 2.

Evaluation.

$Eval(PoV) = 1 + \mathbf{(Number\ of\ successful\ PoVs)/(Number\ of\ competitors - 1)}$

The evaluation term was modified as well, because now a CRS could score against up to six competitors in a single round. Note that a CRS could submit a different PoV against each competitor to target that competitor's replacement binary.

Discussion. It is more difficult to describe possible scoring values in an adversarial environment because security and evaluation scores now depend heavily on the actions of the competitors. For the sake of example, let's suppose there are only two teams in CFE, where Team A plays a specific strategy and Team B does nothing. In that case, we have the following possible score values:

- *Balanced CRS*. If Team A's CRS solves the challenge completely with perfect retention of functionality and performance (and a successful PoV), it would receive a score of $1 \times 2 \times 2 = 4$, while Team B would receive a score of $1 \times 1 \times 1 = 1$.
- *Defense-only CRS*. If Team A's CRS uses a defense-only strategy with perfect retention of functionality and performance, it would receive a score of $1 \times 2 \times 1 = 2$, while Team B would receive a score of $1 \times 2 \times 1 = 2$.
- *Offense-only CRS*. If Team A's CRS uses an offense-only strategy, it would receive a score of $1 \times 2 \times 2 = 4$, while Team B would receive a score of $1 \times 1 \times 1 = 1$.
- *Do-nothing CRS*. If Team A's CRS did nothing, it would receive a score of $1 \times 2 \times 1 = 2$, while Team B would also receive the same score.

The effect of these changes made CFE scores rather different from CQE: the security score would no longer cause a competitor to receive a 0 for a round; in fact, doing nothing (that is, neither replacing a binary nor providing PoVs against competitors) guaranteed a score of 1 if a service was successfully attacked and a score of 2 if no attacks against it were successful. Note that the dynamics between CRSs become much more important in this game—an offense-only strategy works only if the competitor is not expected to find the PoV in the network traffic and turn it around to attack its creator, thus leveling the score.

Competitor Strategies

Given that the intention of the scoring algorithm design was to encourage finding and understanding bugs (by

generating PoVs) and automatically patching vulnerabilities without excessive performance or functionality degradation, what strategies did they select? Were there unintended consequences? In this section, we tackle some of these questions and discuss competitors' strategies in CQE and CFE.

Wall-Time Effects on Competitor Choices

The use of wall time to assess the performance impact of a network filter seemed to affect competitor behavior in at least two ways. First, most competitors avoided the use of network filters entirely; the few filters that were deployed were very simple—they terminated sessions that contained data resembling references to the magic page in attempts to foil Type 2 PoVs. Post-event interviews suggested some teams lacked confidence that they could estimate the performance impact of fielding network filters, and chose to forgo this capability in fear of heavy performance penalties. This finding ran counter to our expectation of teams trying to offload computation into the network appliance, which motivated measuring its performance impact in the first place.

The second behavior that was likely prompted by the use of wall time in the scoring algorithm was the inclusion of infinite loops by one of the teams in their PoVs to degrade the performance of their competitors' services. This team would create PoVs that used remote code execution to enter a tight infinite loop (equivalent of `while (1) ;`) subsequent to scoring (for example, by leaking values of the memory page). This consumes defended host CPU resources and increases the response time of other services on the defended host. We did not perform analysis to determine if the use of this "Type 3 PoV" had a measurable impact on the scores of their competitors. But the fact that at least two teams managed to significantly degrade the performance of their own services by fielding flawed replacement binaries suggests that this strategy has merit.

In CGC, we specifically worked to remove the ability of competitors to cause denials of service by flooding hosted services with traffic. Each CRS could only launch a limited number of PoVs against each service on each competitor's defended host. The Type 3 PoV turned out to be a clever hack to bring down a competitor's score in addition to improving your own; the one downside to this phenomenon was that availability scores for services unrelated to the one being exploited could suffer, which made the process of identifying the culprit for poor performance difficult.

Always Be Throwing

The scoring algorithm imposes no penalties on a team for deploying PoVs against their competitors. Teams could throw PoVs up to 10 times per round against each of their competitors' services; a few teams elected to throw generic PoVs whenever they lacked a working, targeted PoV. One team's generic PoVs simply guessed at a magic page value at random (on one occasion the guess was correct!). However, this team's logic for fielding generic PoVs may have been flawed, because on at least two occasions, their CRS replaced successful targeted PoVs with generic PoVs, which failed to score in subsequent rounds.

From a game-theoretic perspective, there was no disincentive to throw PoVs, and launching attacks (even nonworking ones) provided teams with ability to create additional traffic that would have to be filtered by the network appliance and analyzed by the target team. In addition to using up limited resources, this traffic could be used as a cover for real PoV throws, making them harder to identify.

Zero Score for Consensus Round

The cost of replacing a service in CFE was intended to dissuade teams from relying on frequent replacement as a defensive strategy. Post-event interviews with the teams suggested that this cost led some teams to postpone the submission of replacement binaries and to avoid replacing a binary more than once.

At one extreme, Team A generally only replaced a binary following indication that the associated service was compromised. Their CRS would then not replace that same service again, despite feedback indicating the replacement had zero availability due to performance or functionality degradation. This seemed to be a bug—a post-event interview with Team A indicated that they removed logic that responded to poor availability at the last minute.

Team B employed a different strategy to avoid multiple replacements; their CRS replaced all but three of the services, seemingly independent of whether or not a successful PoV was launched against their services. Team B would never replace that service again, unless the CRS received feedback that the replacement binary had poor availability, in which case the CRS would revert to the original binary. In some instances, Team B's CRS incorrectly reverted services with perfect availability due to a self-inflicted denial-of-service problem caused by heavy CPU usage by an unrelated service running on the shared defended host.

Team C followed this general strategy as well, though they only replaced half of their services, regardless of whether the services had been compromised. Whenever Team C's CRS received feedback that a service had poor availability, it generally uploaded a new replacement binary in response.

From a game-theoretic perspective, the strategies chosen by these teams seem suboptimal. Ideally, a CRS should have enough situational awareness to determine when a successful PoV against their service has been launched and replace the binary with a patched version at that point. Preemptive replacement of services unnecessarily cost several teams points due to the consensus round downtime, even though no successful PoVs were fielded against these services in CFE.

Defenses in CQE versus CFE

During CQE, teams knew that a working PoV would be deployed against each of their replacement binaries, which motivated the deployment of defenses—in fact, if no reference PoVs were mitigated, a team would receive a score of 0 on that submission. In CFE, reference PoVs were not deployed against the services, and any successful attacks would have to come from competitors. Thus, a team's decision to deploy a defense might depend on whether they believed a given service could be compromised by one of their competitors.

In a PoV-rich environment, where many services are proven vulnerable, a good strategy (one achieving the highest score) might be to focus on patching and patch preemptively, as it might prevent a round of reduced score when the service is proven vulnerable. However, in a PoV-limited environment, where many services are not proven vulnerable, a good strategy would be to patch only in response to a successful attack, as discussed earlier. In CFE, only 20 challenge sets out of 82 were proven vulnerable by competitors, so a strategy of patching in response to an attack would have produced higher scores. The use of reference PoVs in CFE might have provided additional motivation to field defenses and enabled better demonstration of CRS patching capabilities observed in CQE.

Point Patches versus Generic Patches

While the design of the competition was not intended to preclude the use of any given defensive strategy, we did strive to encourage correcting the program flaws rather than providing generic mitigations that simply masked the presence of those flaws. We did not feel that liberal application of control flow integrity techniques would advance the state of the art of network defense. The availability element of the scoring algorithm was designed in part to reward targeted patches by penalizing increased memory use and CPU execution time.

However, as can been seen in Figure 2, many replacement binaries were fielded that consumed significantly fewer resources than the associated reference services. Figure 2 shows ratios of resources consumed by replacement binaries divided by resources consumed by reference binaries, averaged over each round. Each color in the figure represents a different team, and the different services are arrayed along the *x*-axis.

The ability of a replacement binary to consume fewer resources than the reference service is an unintended effect resulting from the unforgiving performance scoring function combined with the fact that CGC challenge binaries were simple services and thus much smaller than most real-world network services. Some utilized a very small number of pages of memory, and thus increased memory usage of even a single page might drive the availability score to zero. CPU execution times of very short-running services presented a similar problem because any modest increase in processing time resulted in large proportional increases in the measurement of CPU cycles.

To mitigate this problem of very small challenge binaries, the CGC build process deliberately included extra data (in the form of an embedded PDF) and processing (in the form of a CRC computation across the PDF). Neither extra data nor extra computation was necessary for correct service functionality; therefore, both could be safely removed from each binary. During the testing phase before CFE, it was possible for teams to learn that space and time cushions could be obtained by removing the CRC-related code and the PDF data. As a result, teams were able to deploy generic defenses without incurring significant availability costs. This experience indicates that availability constraints alone cannot effectively mandate the use "point patches" rather than generic defenses on small programs.

Effect of Consensus Evaluation

One of the motivations for consensus evaluation was to afford teams the opportunity to find bugs inadvertently introduced by competitors' patches. We found no evidence of any team successfully exploiting a vulnerability unintentionally introduced by another team. We did note two instances of intended insertions of vulnerabilities, motivated by the knowledge that competitors could analyze patched binaries. The first insertion was a honeypot that one team included in most of their replacement binaries. This honeypot was a simple buffer overflow that was easy to find and exploit, but could not be reached when the service was executing on CGC Final Event hardware as a result of an execution divergence between CGC hardware and common analysis environments due to handling of `cpuid` instruction. This honeypot caused several opposing teams to field PoVs designed to exploit the unreachable flaw, thus providing an effective security countermeasure.

The second type of deliberate flaw was a back door embedded in a replacement binary, intended to be

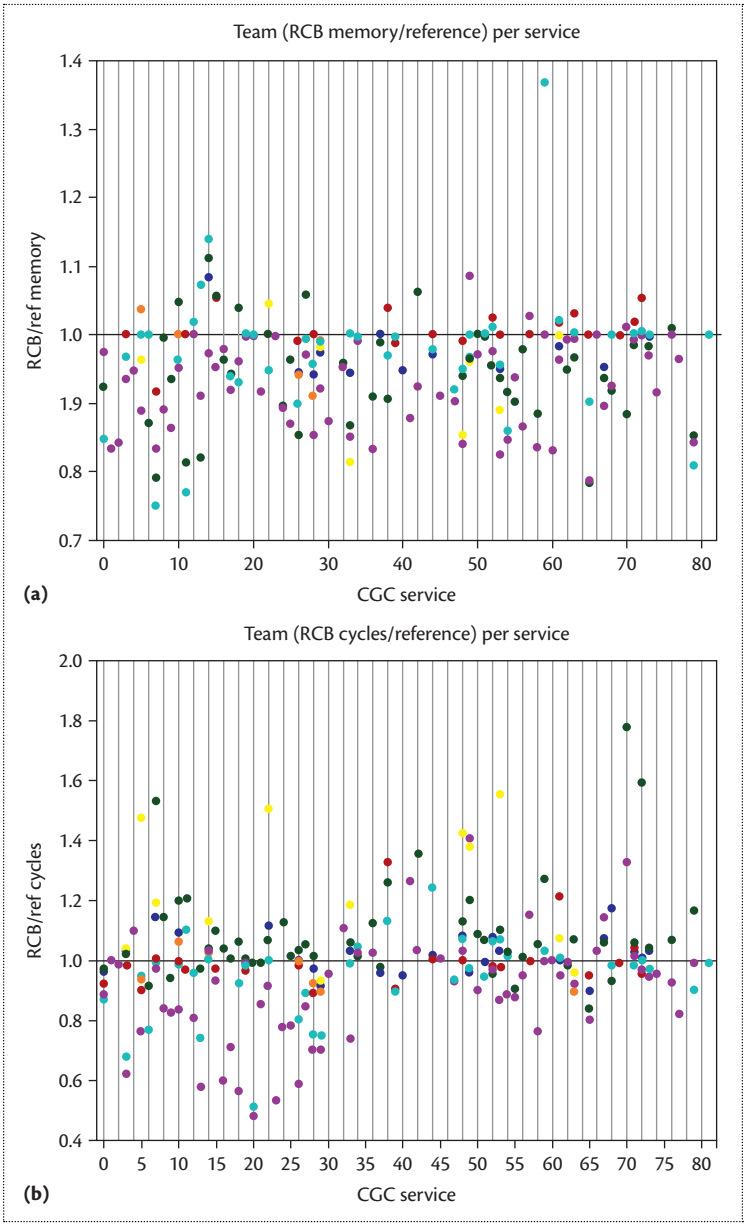


Figure 2. Average (a) memory use and (b) execution time factors for CFE replacement binaries. Each point represents average memory use or CPU execution time factor for replacement binaries by a particular CRS for a particular service.

exploited if another team elected to utilize this binary as their own replacement for the backdoored service. Post-event interviews indicated that multiple teams inserted back doors into their replacement binaries; however, no backdoored service was redeployed during CFE. This behavior was anticipated by CGC organizers; earlier in the project, some teams expressed concerns that their replacement binaries might get reused by the competitors who would be effectively "free-riding"; adding a back door accessible only to the team that created the patch dissuades such behavior.

Lessons Learned

Designing and running any capture-the-flag event is a nontrivial undertaking; organizing a high-profile completely automated capture-the-flag event is doubly so. Despite best efforts on the part of the game designers, some things do not go according to plan, and unintended consequences of scoring or measurement decisions can drive competitors to nonoptimal strategies. In this section, we review several lessons learned as part of organizing and running the CGC Qualifying and Final Events.

Red-Team Scoring Algorithm

When designing the scoring algorithms for CQE and CFE, we spent much time considering competitor strategies that would achieve good scores but not advance the state of the art in automated network defense. In several cases, we had to revise the scoring algorithm to provide disincentives for such strategies (for example, requiring that at least one reference PoV is mitigated in CQE or assigning a significant penalty for service replacement). Much of this effort also focused on discouraging collusion between teams. When performing such analyses, it is useful to consider teams that might "do nothing," teams that might play defense only, offense only, or some sort of balanced or randomized strategy. Each team persona might illuminate a different corner of the scoring space and provide ideas for improving the scoring and redirecting competitors away from undesirable strategies.

Make Measurements Reproducible

Any scoring algorithm is closely tied to the measurements that support it; in case of CGC, the measurements included functionality, wall time, CPU execution time, memory usage, and whether a PoV thrown against a challenge binary successfully proved vulnerability. When designing the measurement framework and scoring algorithm, we focused on ensuring reproducibility of scores; that is, running the same round multiple times with the same inputs should produce the same scores for the competing teams. This meant that the game infrastructure had to go through great pains to control the use of randomness in the game (all randomness available to the challenge binaries and PoVs was an output of a pseudorandom number generator with a known seed) and limit effects of other nondeterminism sources: process scheduling, networking issues, and so on. When a particular measurement could not be contained to an acceptable level of variance due to these sources of nondeterminism, the scoring algorithm had

to be adjusted to contain the effects from such measurements on the resulting score.

Make Metrics Transparent

Much of the scoring algorithm design for CGC was driven by the maxim that "you get what you measure." By measuring additional memory usage and additional CPU cycles, we hoped to get patches that focused on the flaw rather than general program-hardening techniques. However, this strategy really only works if the competitors understand what exactly is being measured. Instead of providing a clear statement that we would measure CPU cycles consumed while the process executed in user space, we provided a series of oracles that the teams could interact with to divine the effects of techniques embedded in their replacement binaries on availability. Prior to CQE, this was a sequence of "scored events," and for CFE, this was the "sparring partner." Part of the reasoning for this choice was to prevent the competitors from gaming the scoring mechanism; however, it resulted in too much ambiguity and caused some competitors to model the performance metrics incorrectly.

Avoid Wall Time as a Metric

Our choice to use wall time as a metric to measure availability costs of network filters substantially complicated implementation and testing of the CGC game infrastructure. One problem is that services that complete relatively quickly become very sensitive to small variations in wall time. Thus, differences in kernel scheduling or TCP networking effects could result in significant variations between otherwise identical sessions. This complicated our ability to achieve repeatable results, which are necessary when establishing a baseline against which to measure the performance of replacement binaries and network filters.

We investigated an alternative to wall time by measuring aggregate CPU cycles on the network appliance component for all polls of a given service. However, when first attempted, the network appliance was hosted on Linux, and the CPU cycle measurements had very high variations. We were finally able to achieve repeatable wall-time measurements after converting all of the infrastructure components to FreeBSD and carefully tuning the kernel configurations.

In this article, we presented our experience and lessons learned in designing and implementing the scoring algorithms for the CGC Qualifying and Final Events. These algorithms succeeded in incentivizing competitors to develop systems that could automatically patch previously unseen binaries to mitigate vulnerabilities as well as provide proofs of vulnerabilities for these binaries. The scoring algorithms in CGC were designed for automated evaluation and strived to achieve fairness, collusion resistance, and real-world relevance, and in many aspects we believe that they succeeded. We hope that knowledge of our experience proves useful to other capture-the-flag competition designers and helps them avoid some of the pitfalls we faced. ■

Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US government.

References

1. "Cyber Grand Challenge: Rules," Version 3, DARPA; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_Rules_18_Nov_14_Version_3.pdf.
2. "Cyber Grand Challenge: Frequently Asked Questions (FAQ)," DARPA; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_FAQ.pdf.
3. "Proof of Vulnerability Markup Language," DARPA; https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/cfe-pov-markup-spec.txt.
4. "The BlueHat Prize Contest Official Rules," Microsoft; https://web.archive.org/web/20111120054734/http://www.microsoft.com:80/security/bluehatprize/rules.aspx.

Benjamin Price is a member of the Cyber Analytics and Decision Systems Group at MIT Lincoln Laboratory. Email at ben.price@ll.mit.edu.

Michael Zhivich joined MIT Lincoln Laboratory in 2005 as a member of the Secure Resilient Systems and Technology Group. Email at mzhivich@gmail.com.

Michael Thompson is a research associate at the Naval Postgraduate School in Monterey, California. Email at mfthomps@nps.edu.

Chris Eagle is a senior lecturer of computer science at the Naval Postgraduate School in Monterey, California. Email at cseagle@nps.edu.

A Honeybug for Automated Cyber Reasoning Systems

Timothy Bryant and Shaun Davenport | Raytheon

From the editors: While all the articles in this issue share the special issue theme, two are uniquely related in that they present differing perspectives of one particular strategy employed in the Cyber Grand Challenge Final Event. In "A Honeybug for Automated Cyber Reasoning Systems" the authors (creators of the Rubeus system) detail an approach aimed to thwart the anticipated analysis techniques of other competitors. Contrarily, "Effects of a Honeypot on the Cyber Grand Challenge Final Event" provides an analysis of this very approach from the perspective of a contest referee—a member of the competition framework integrity team. Between the two perspectives, one sees the approach's reasoning and potential risk as well as an unbiased analysis of the implementation and an account of its efficacy on the other competitors.

In the Cyber Grand Challenge, our cyber reasoning system patched binaries with a honeybug that enticed competitor systems to pursue the honeybug instead of exploiting actual vulnerabilities. Methods for detecting and countering automation in the real world may be used to thwart the ability of malicious actors to find vulnerabilities.

"The enemy knows the system" was an essential design principle for DARPA's Cyber Grand Challenge. Also known as Shannon's Maxim,[1] it ensured that competing automated cyber reasoning systems (CRSs) could evaluate the effectiveness of each other's defenses. A cyber reasoning system automates the search for vulnerabilities and applies appropriate patches to protect the software from future attacks. In the Cyber Grand Challenge, CRSs did not analyze black boxes across a network, but instead analyzed competitor binaries disclosed to them by the game infrastructure.

A honeybug is a pseudo-vulnerability planted in a binary with the goal of diverting the attention of a CRS away from legitimate existing vulnerabilities. Our honeybug patch was a strategy used by Rubeus, our CRS, to subvert our enemies' ability to "know the system." Although we could not prevent the distribution of our binaries, we could perhaps trick competitors if our binaries behaved differently on their systems than they did on the official scoring system. If so, it would be possible to prevent competitor systems from searching for legitimate proofs of vulnerabilities (PoVs) in our binaries or prevent them from validating legitimate PoVs against our binaries.

Moreover, if our binaries behaved differently on competitor systems, we could manipulate those systems because they would be interacting with data that we controlled (that is, our patched binaries). Detecting emulated or virtualized environments is a well-studied problem and various techniques have been developed[2]—and human teams playing capture-the-flag (CTF) games have used shenanigans to gain an advantage over other teams.[3] In contrast, the intent of our honeybug was to build into our CRS the ability to detect other cyber reasoning systems and to manipulate them.

 Copublished by the IEEE Computer and Reliability Societies

Enabling Shenanigans

Ironically, this strategy was made possible because the designers of the Cyber Grand Challenge afforded themselves the dubious benefit of "security through obscurity" by using an undisclosed platform for scoring. While our binaries were to be fully exposed to competitors, the teams themselves had little visibility into the execution environment of the scoring system.

The DECREE (DARPA Experimental Cybersecurity Research Evaluation Environment) operating system (https://github.com/CyberGrandChallenge), with its carefully circumscribed set of seven system calls, removed the ability to securely bind the execution of our binaries to our designated host. In addition, DECREE prohibited high-fidelity timing information (for instance, RDTSC instruction), which can often be used to detect different execution environments. Thus, for many months we assumed there was no mechanism available to our binaries for detecting whether or not they were running on the official scoring system.

A couple months before the final event, we considered the CPUID instruction that reports details about the underlying processor. We could use it to identify the official scoring system, but only if the processor used by the scoring system was both unique and generally unknown to the rest of the competition. Otherwise, teams would certainly update their CRS's model of the execution environment to match that of the official scoring system. In addition, we recognized that because our CRS was allowed to monitor the network traffic to and from our binaries running on the scoring system, we could exfiltrate the CPUID values from our binaries back to our CRS by embedding those values in the data that our binaries transmitted.

A second enabler of our strategy was that our competitors were fully autonomous machines. This meant that no humans would be scrutinizing the CPUID check, recognizing its purpose, and removing the check from our binary. To avoid falling for this trick, a CRS must understand the purpose of the code—that normal functioning occurs on the official scoring system (or a system with identical CPUID values) and that alternative code is executed otherwise. Unless the other teams had preprogrammed their CRSs to recognize and react to the CPUID check (or they had correctly modeled the scoring system's execution environment), there was a strong chance that their CRSs would fall for the trick.

Fingerprinting the Scoring System

Because this technique was predicated on secrecy, we wanted to quietly fingerprint the scoring system. However, we didn't have access to the scoring system until the testing began just a few weeks prior to the final event. During the testing phase, our CRS competed against mock sparring partners to verify that our system was operational and ready for the final event. If, during these sparring sessions, the CGC organizers discovered that we were using the CPUID, they could decide to change the values reported by the scoring system before or even during the final event. Alternatively, they might publish the CPUID values and guarantee to all teams that those values would remain fixed. Such actions would prompt our competitors to scramble to update their CRS to properly model this aspect of the scoring system.

So, we submitted a single patched binary to the scoring system during one of the test runs. Instead of submitting a correctly secured binary, the purpose of this special binary was to query a large range of CPUID values and to transmit the values as data over the network to be picked up by our CRS that was monitoring the network traffic. As the day of the competition drew near and the CGC team indicated that the scoring system code was tested and "frozen," we worried less about being discovered, and our exfiltration of the CPUID values from the scoring system even included the playful plaintext phrase "NOT FOR YOUR EYES."

> **"The intent of our honeybug was to build into our CRS the ability to detect other cyber reasoning systems and to manipulate them."**

We later learned that the infrastructure team observed our fingerprinting activities just a few days before the final event during a monitored test run, and it prompted the team to fix a bug in their visualization system. They would soon be projecting their visualizations of running programs on large screens in front of a live audience. To perform these visualizations, this part of their system replicated the execution of competitor binaries in an emulated environment. However, that environment was not identical to the scoring system. In particular, it reported different CPUID values. Thus, they had to fix the CPUID values being reported by the emulator. Otherwise, the audience would have seen the honeybug execution path instead of the actual execution occurring on the scoring system.

The fingerprinting results were quite perplexing. The scoring system claimed it was using an Intel i5-4258U processor—a common processor for laptops, not a server with thousands of CPU cores. The scoring system environment was reporting additional CPUID

values outside the range of what Intel x86 processors are supposed to report. Moreover, it was reporting values that were different from commonly used emulators and hypervisors. We were able to conclude that the scoring system was either heavily modified or an entirely custom emulator or hypervisor. We settled on 12 CPUID values that best differentiated the scoring system from other emulators and hypervisors that were likely to be used by our competitors.

Of course, we had to ensure that our CRS correctly modeled the CPUID values of the scoring system. Otherwise, it would be susceptible to the very trickery we hoped to use against our competitors. During the final event, the fingerprinting patch checked if the current CPUID values reported from the scoring system had changed. If the values had indeed changed, the patch would sacrifice our availability score to report back the correct values (that is, the patched binary could either exfiltrate values from the scoring system or respond correctly to the functionality tests, but not both at the same time). Thus, instead of transmitting the proper response, our binary transmitted the CPUID values. Because our CRS monitored the network traffic to and from our running binaries, it would pick up the new values and distribute them to the rest of the system, updating our hypervisors and patching subsystem accordingly. Although this never occurred during the final event, our CRS was prepared.

> **We made the bug so obvious and inescapable that only a machine would not stop to ask why it was there.**

Also, if the reported values changed too often (we used a conservative threshold of more than 1 for the final event), our CRS would cancel the whole scheme. Not only would it not be able to employ any shenanigans, but neither would any other CRS. Therefore, it would no longer need to update its model of the scoring system.

Choosing the Effect

Now that our binary could detect when it was running in a competitor CRS's environment (that is, anywhere but the scoring system that we had fingerprinted), we wanted to leverage this capability to our greatest advantage. We considered various options including denial of service, evading analysis, and remote code execution.

At the time, we did not consider a Rowhammer-style attack[4] to disable competitor systems. This style of attack manipulates DRAM memory cells to corrupt memory regions that are otherwise protected by the operating system against unauthorized access. Because all of the CRSs ran on identical hardware and we knew the exact hardware specifications, and because we could also reasonably guess the hypervisors and emulators used by our competitors, this attack may have been feasible. However, disabling a competitor's CRS using this method would have violated the official rules and participation agreement.[5,6]

One idea that we considered was making the patched program run in a computationally intensive infinite loop. Although it would be beneficial to consume competitors' analysis resources, we figured that most CRS implementations would enforce timeouts and that it would probably not have a huge impact on systems with 1,024 CPU cores. We also considered using a simple terminate syscall—prematurely ending execution to evade dynamic analysis. Although this may prevent a vulnerable CRS from locally validating a legitimate PoV against our binary, PoVs that were already submitted would continue to be used for scoring unless they were explicitly replaced. Consequently, we turned our attention to manipulating competitor CRSs into unwittingly submitting invalid PoVs.

Our honeybug was a planted vulnerability that we wanted CRSs to easily find and prove. Specifically, we implemented a simple stack buffer overflow that only required 12 bytes of input to overwrite the return address and thereby gain code execution. We also ensured CRSs could readily control both the program counter and a general-purpose register (control of both was required for a Type 1 PoV). We made the bug so obvious and inescapable that only a machine would not stop to ask why it was there. We hoped, however, that CRSs were sophisticated enough to actually go about the business of analyzing our patched binary.

The mechanics of patching binaries with the honeybug were straightforward. Our CRS deployed the honeybug opportunistically—only when it was already patching a bug or adding a defensive mitigation—because the penalty for patching was high. The CRS also ensured that the honeybug patch would always execute before our other defensive patches such as address randomization (ASLR) or nonexecutable stack (NX).

Gambling with a Honeybug in Vegas

Planting a honeybug in our binaries was not without risk. Backdoors that rely on secrecy are often discovered and reduce security. Although our honeybug is not a traditional backdoor, it could make some vulnerabilities

easier to prove. For example, suppose a CRS found an existing vulnerability where it controlled the program counter (`eip`) but not a general-purpose register. Normally, such a bug would not be exploitable. However, if we insert the honeybug, an intelligent adversary can then exploit the original bug by simply redirecting code execution to our honeybug, thereby gaining control of `eip` and a general-purpose register and proving the vulnerability that we introduced! If our CRS were competing against humans, we would not choose to use a honeybug because skilled human CTF players would not fall for the CPUID trick and they would readily take advantage of the planted vulnerability.

Against machines, we gambled that the designers of the machines had presupposed that reference (unpatched) binaries are always easier to exploit than patched binaries. The reasoning is that patched binaries are likely to include anti-analysis and exploitation mitigations. Even if some patches are not completely effective, they would be unlikely to intentionally introduce new vulnerabilities. Therefore, we expected that CRSs would not look for novel vulnerabilities introduced by our patches.

Furthermore, in a multiplayer game with several CRSs issuing patches, there could be many versions of the same binary to analyze. For efficiency, we expected CRSs to allocate their computational resources to finding vulnerabilities in the reference binaries and only afterward adapt those PoVs to patched binaries.

Honeybug Enticement

We also wagered that CRSs would behave systematically, indifferently, and greedily. The common expectation was that challenge binaries would contain few vulnerabilities, perhaps even just one. We believed competitor systems would be designed to readily act on any bug discovered. Thus, if the honeybug forced a program to crash at the very beginning of its execution, a CRS would detect this crash, construct a corresponding PoV, and submit the new bogus PoV.

Our honeybug targeted machines that analyzed our binary or machines that actively verified and adapted PoVs that they had already discovered in the unpatched binary. We didn't expect the technique to work against all CRSs. If it worked against all CRSs, it would be a simple and complete defense. To increase our security score, the honeybug would need to entice all CRSs that were currently throwing a valid PoV. Although this scenario was unlikely, we did expect to manipulate some of the CRSs and decrease their evaluation score.

Interestingly, a less sophisticated CRS is less susceptible to this trickery. For example, a simple CRS may submit PoVs that it finds against a reference binary without first validating that they work against patched binaries. Another example is a CRS that does not or cannot adapt PoVs it finds in reference binaries to work against our honeybug patched binary. Because the CRS does not submit a different PoV, the scoring system continues to use the previously submitted PoV. Thus, this CRS is unaffected by the honeybug.

Table 1. Cyber reasoning systems submitting PoVs to exploit the honeybug.

Cyber reasoning system	# of honeybug PoVs scored	# of unique challenges
Jima	96	27
Mayhem	65	7
Galactica	31	4
Mechaphish	9	2
Crspy	2	1
Xandra	0	0

Results

We examined the results[7] from the Cyber Grand Challenge final event to determine how frequently competitor CRSs attempted to exploit the honeybug. It is difficult to assess the extent to which our honeybug impacted our final placement, because without knowing the internals of competitor CRSs, we can only speculate about possible adverse impacts to their overall performance. Moreover, the potential of the honeybug to alter overall scores depends in part on how many challenges for which CRSs are able to find proofs of vulnerability. Nevertheless, we were able, after the competition, to evaluate each PoV that was submitted against our patched binaries containing the honeybug. For each round where this occurred, we ran the competitor's PoV 10 times using random seeds. We configured our test environment to report CPUID values that differed from the official scoring system to force execution to the honeybug. Because the honeybug is vastly different from the real vulnerabilities in the CGC corpus, a successful PoV in our test implies that the honeybug was being targeted by the competitor. We ran this test for all rounds of the final event and tallied the results in Table 1.

Beyond the Cyber Grand Challenge

Is our honeybug a mere curiosity, taking advantage of a unique discrepancy between the Cyber Grand Challenge scoring system and the automated cyber

reasoning systems? Or, does it actually point to something more relevant to the security community? Our honeybug illustrates that automated systems may need to reason about whether or not software is acting deceptively. It also clearly shows that cyber reasoning systems may be susceptible to manipulation. For CGC, the honeybug detected competitor CRSs by proxy using the execution environment. Perhaps, however, detecting the execution environment is not the only way to detect and manipulate a CRS.

Let's consider one important example. Today, most vulnerabilities in binaries are still discovered by fuzzing tools (many of which can be considered a primitive CRS). In fact, many CRSs rely heavily on fuzzing to produce a majority of the vulnerabilities they find,[8–10] and yet we are unaware of any software today that intentionally thwarts fuzzing-based tools. Fuzzing typically requires executing millions of test cases before a vulnerability is discovered, and many systems targeted by malicious actors are fuzzed without resetting the state of the system between each test case. This happens when there is no suitable emulator for the system, or because there is no ability to rapidly reset the system to a clean state between test cases.

Software that is permitted to retain state while being fuzzed (a condition that was not satisfied for the Cyber Grand Challenge) could differentiate between a fuzzing campaign and legitimate use. Password-based authentication mechanisms have long used the strategy: too many illegitimate attempts and either you lock yourself out or a significant delay is imposed on subsequent attempts. In a similar manner, one can imagine software that detects a fuzzing campaign and that imposes delays or redirects execution to dead ends or false bugs.

The mechanism for detecting and diverting a fuzzing campaign could be made difficult for humans to reverse engineer, let alone automated reasoning systems. Devising an efficient and effective defensive mechanism of this sort would be an interesting line of research. CRSs could continue to be used by software vendors in their development processes to find vulnerabilities before the software is shipped by using builds that do not contain the protections. However, the shipped software with the protections in place would reduce the ability of others to use CRSs to find vulnerabilities for malicious purposes, at least for some categories of software products.

Our honeybug demonstrates that cyber reasoning systems, like other software, may be susceptible to manipulation. In our example, the detection mechanism leveraged a difference between a CRS's model of the execution environment and the real execution environment. Once detected, our honeybug was able in some cases to prevent further analysis of our binaries and to provoke competitor systems to submit bogus proofs of vulnerability. The honeybug was a small but intriguing component of the Rubeus cyber reasoning system. ■

References

1. C. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28, no. 4, Oct. 1949.
2. P. Ferrie, *Attacks on Virtual Machine Emulators*, white paper, Symantec Corporation, Jan. 2007; http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.
3. B. Schmidt and P. Makowski, "Dissecting Wireshark: A Case Study on Network Anti-Forensics," OSDFCon, 2014; https://www.osdfcon.org/presentations/2014/Schmidt-OSDFCon2014.pdf.
4. Y. Kim et al., "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors," *Proc. 41st Int'l Symp. Computer Architecture* (ISCA 14), 2014; doi:10.1109/ISCA.2014.6853210.
5. "CGC Frequently Asked Questions," DARPA, Aug. 2016; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_FAQ.pdf.
6. "CGC Participation Agreement," DARPA, 16 May 2014; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_Extended_Application_form_v5_16_May_2014.docx.
7. B. Caswell, "Cyber Grand Challenge Corpus," 4 Jan. 2017; http://lungetech.com/cgc-corpus.
8. S.K. Cha et al., "Unleashing Mayhem on Binary Code," *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.
9. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," NDSS, 2016.
10. "American Fuzzy Lop," coredump; http://lcamtuf.coredump.cx/afl.

Timothy Bryant is a principal engineer at Raytheon developing automated vulnerability assessment technology. He led Raytheon's Deep Red team in the Cyber Grand Challenge. His research interests include dynamic and static analysis of binaries and artificial intelligence applied to computer security. Bryant has an MS in computer science from Columbia University. Contact at timothy.k.bryant@icloud.com.

Shaun Davenport is a vulnerability researcher at Raytheon. His interests include static analysis and embedded device vulnerabilities. Davenport studied computer science at Florida Institute of Technology. Contact at shaungdavenport@gmail.com.

Effects of a Honeypot on the Cyber Grand Challenge Final Event

Michael F. Thompson | Naval Postgraduate School

From the editors: While all the articles in this issue share the special issue theme, two are uniquely related in that they present differing perspectives of one particular strategy employed in the Cyber Grand Challenge Final Event. In "A Honeybug for Automated Cyber Reasoning Systems" the authors (creators of the Rubeus system) detail an approach aimed to thwart the anticipated analysis techniques of other competitors. Contrarily, "Effects of a Honeypot on the Cyber Grand Challenge Final Event" provides an analysis of this very approach from the perspective of a contest referee—a member of the competition framework integrity team. Between the two perspectives, one sees the approach's reasoning and potential risk as well as an unbiased analysis of the implementation and an account of its efficacy on the other competitors.

During the Cyber Grand Challenge Final Event, one team included a honeypot within most of their "patched" services, fooling some competitors into abandoning working proofs of vulnerabilities to instead attack the honeypot. However, logic within the patched service appears to have contained a flaw, which led to the service crashing.

The Cyber Grand Challenge (CGC) Final Event (CFE) included "consensus evaluation," in which, whenever a cyber reasoning system (CRS) submitted a replacement service (for instance, to mitigate potential vulnerabilities in the service), the team's competitors were provided a copy of the replacement service prior to the service being fielded. One intent of consensus evaluation was to allow competitors to determine if vulnerabilities were in fact mitigated or simply masked by a reformulation of the service. For example, shifting a stack address might defeat a proof of vulnerability (PoV) crafted for the original version of the service, but it could be compensated for through analysis of the revised service. Consensus evaluation also provided the competitors an opportunity to determine if a patch may have inadvertently introduced a new flaw into the service, and it led some teams to deliberately introduce flaws in to their services.

A team may have chosen to introduce an intentional flaw for one of two reasons. The first reason, broadly anticipated, would be to fool another competing team into adopting their "patched" service. Suppose Team B saw Team A release a new version of service X. Team B may conclude that this version of service X mitigates a vulnerability in the service, leading Team B to deploy that same version. At that point, Team A could then exploit the vulnerability that they themselves had inserted into service X. This strategy of inserting a back door into a service was described by the Shellphish team.[1] The second, less anticipated form of deliberate flaw is a honeypot intended to lure competitors away from pursing actual vulnerabilities.

One of the CFE teams, Deep Red, designed their CRS, Rubeus, to embed a honeypot in most of the replacement challenge binaries (RCBs) that they fielded. The honeypot contained a simple buffer

1540-7993/18/$33.00 © 2018 IEEE Copublished by the IEEE Computer and Reliability Societies March/April 2018

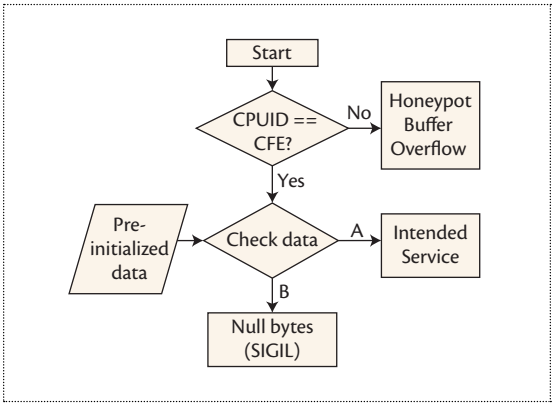


Figure 1. Three paths from the honeypot initial control flow.

overflow vulnerability reachable only while the binary was not executing on the CGC game infrastructure hardware. Several competitor systems created PoVs that targeted this honeypot. And in at least three cases, competitor systems replaced PoVs that were successful in previous rounds with PoVs that targeted the honeypot, but were impotent against the actual vulnerability they had successfully exploited with the abandoned PoV. Had these competitors left the incumbent PoV in place, they would have continued to score.

This article describes the mechanics of the honeypot and the manner in which it was deployed. I identify several cases in which Rubeus's deployment of RCBs appeared arbitrary and detrimental to their score in CFE. Flaws related to their honeypot deployment logic resulted in at least 37 of their 1,422 service rounds having a score of zero. I also describe cases where the honeypot was successful in both diverting the attention of competitors and causing competitors to abandon working PoVs.

In this article, the word *competitor* refers to the CRS built by each contestant team to autonomously analyze, patch, and prove software vulnerabilities in CGC Challenge Binary services (CBs). CFE CBs are identified by their CGC identifiers, for instance, "CROMU_00065."

Mechanics of the Honeypot

The honeypot created by Deep Red and deployed by their CRS, Rubeus, utilized divergent execution to ensure the service was not vulnerable to the honeypot's buffer overflow when executing on the CGC game infrastructure. The honeypot used the IA32 *cpuid* instruction[2] to obtain information about the executing binary's host processor. The CGC DECREE execution environment employed a hypervisor during CFE that caught the *cpuid* instruction and returned a result that was constant. In other words, a binary that executed the *cpuid* on CGC game infrastructure hardware would always get the same result as provided by the hypervisor. However, if the DECREE execution environment ran on hardware or a virtual machine that lacked the CGC hypervisor, then the *cpuid* instruction returned a result that depended on the underlying hardware rather than the constant value generated by the CGC hypervisor. Teams could have deployed more robust analysis environments that would not diverge on the *cpuid* check. As illustrated in Figure 1, the honeypot branched on the *cpuid* results, executing the CB service on game infrastructure, and code vulnerable to the honeypot's buffer overflow on all other systems. However, there was a third branch, labeled in Figure 1 as "B," leading from the Check data function. This branch led to execution of null bytes and thus crashing of the service. As will be seen below, this branch was taken many times while executing on CFE infrastructure.

Leading up to the CFE, teams were provided access to the CFE game infrastructure during "sparring sessions"[3] in which each CRS could compete against a simulated adversary, test their APIs, and experiment with strategies. The CGC competition framework developers anticipated that some teams would use this opportunity to fingerprint the underlying hardware platform. Deep Red apparently used a sparring session to record the results of the *cpuid* instruction, enabling their RCBs to potentially distinguish CGC game infrastructure from other platforms.

Rubeus's Deployment of RCBs

The CFE included 82 distinct services, of which 20 were exploited in the course of the game. Deep Red deployed one or more replacements for 32 of the 82 services. Rubeus deployed multiple versions of some services, in different rounds. Of the 32 services that Rubeus replaced, 28 contained the honeypot in at least one round.

The CFE logs do not suggest much in the way of coherent strategy in Rubeus's fielding of RCBs. In general, services for which Rubeus never deployed an RCB were never scored on. (One CB was withdrawn from the game before Rubeus could deploy an RCB in response to it being scored on.) The converse is not true. Rubeus deployed RCBs for 26 services that were never successfully exploited either by Rubeus or their attackers. For 14 of these 26 services, they deployed multiple different RCBs, in one case (CROMU_00095) churning four different replacements for a service that was never exploited. Note that each time a team replaces a service, that service is down for a full round, giving the team zero points for that service round.[3]

All the RCBs that contained the honeypot also included a few other generic defensive strategies, the

most significant of which was an allocation of a new stack that was not executable (NX),[4] with a randomized base address. Other generic defensive strategies used by Rubeus included retrieving syscall numbers from data, presumably to complicate static analysis, and using a jump in the start function rather than a call/return. With a few exceptions, the Rubeus RCBs contained no other defenses or patches. Most of the RCBs were accompanied by an intrusion detection system (IDS) filter, whose function was to detect and adjust references to the protected memory page used for a type of PoV that demonstrated an ability to disclose selected memory content. Rubeus may have withheld the filter in the few exceptions because of either performance impacts or tests indicating the filter would degrade functionality of service polls.

For several services, the initial RCB deployed by Rubeus was later followed by another change to the service. However, these additional RCBs usually did not include any defensive additions or patches. Rather, they included changes to the initial data that controlled whether the execution path proceeded to the honeypot or the actual service. As noted above and illustrated in Figure 1, this logic included a third execution path, which resulted in an immediate illegal instruction fault when followed. The second version of an RCB deployed by Rubeus for several services took this fatal execution path for about four rounds (one round to run, the next round to notice trouble and submit a replacement, a round down for consensus, and the final round to deploy the replacement). I could not determine the intent of this third path, though it may be related to functionality that Deep Red alluded to in a postevent interview in which they described a strategy for determining if the expected *cpuid* result had changed between sparring and CFE.

Why All the Replacement Binaries?

One possibility is that the Rubeus RCB churning resulted from their CRS reacting to attacks against their honeypot. Their CRS did appear to deploy RCBs in response to successful exploits, and their pattern for doing that was not readily discernable from their deploying a different RCB subsequent to an attack on the honeypot. The CGC game infrastructure provided competitors with feedback reflecting whether their services were exploited. This feedback would have reflected successful defense of (or more accurately, the lack of a successful exploit against) the RCBs in question. However, each competitor CRS also received a copy of all network traffic directed at their services, and competitors could ingest this traffic into instrumented instances of their services, for instance, in attempt to detect PoVs. If the Deep Red design deployed these instrumented service instances on platforms that respond "incorrectly" to the *cpuid* instruction, the CRS might conclude that the RCB had been exploited. In other words, Rubeus might have incorrectly detected opponent PoV success and therefore reacted inappropriately. However, I found no clear evidence that they were reacting to attacks against the honeypot.

Another posited theory behind their RCB churning assumes flaws in logic handling responses to reduction in availability. For example, if one rogue RCB caused degradation in the availability of other RCBs fielded by Rubeus, the CRS might replace those other RCBs with instances measured to have better performance characteristics. While Rubeus did in fact deploy one RCB that seriously degraded scores of other RCBs for two rounds, most of the Rubeus RCB churning was preceded by rounds in which the Rubeus services had perfect or high availability. The CFE logs show no correlation between degraded availability of Rubeus services and their churning of RCBs.

Effectiveness

Only one competitor avoided the honeypot entirely. Two competitors abandoned perfectly good PoVs in pursuit of honey. And the CFE champion attacked the honeypot many times and never did score against an RCB containing the honeypot. Pursuit of the honeypot by Rubeus's competitors did not appear to alter the final team rankings. Determining the quantitative effect on team scores would require speculation as to the effectiveness of PoVs that may have been thrown in place of those thrown against the honeypot, and further speculation as to how Rubeus would have responded to those attacks. To put some of the quantities presented below into context, the CFE had 92 rounds and 82 different challenge sets, with about 15 challenge sets active in each round.

Xandra

Xandra never pursued the honeypot and scored on two Rubeus RCBs containing the honeypot.

Galactica

Galactica abandoned a successful PoV against NRFIN_00063 and, instead, attacked the honeypot for eight rounds. They also abandoned a working PoV against CROMU_00051 and attacked the honeypot for five rounds. Their working PoV for CROMU_00051 would fail against the Rubeus RCB because that RCB included a randomized NX stack as part of its generic defenses. However, the PoV did cause the RCB to execute a *retn* using a PoV-provided address, which could have led to exploitation of

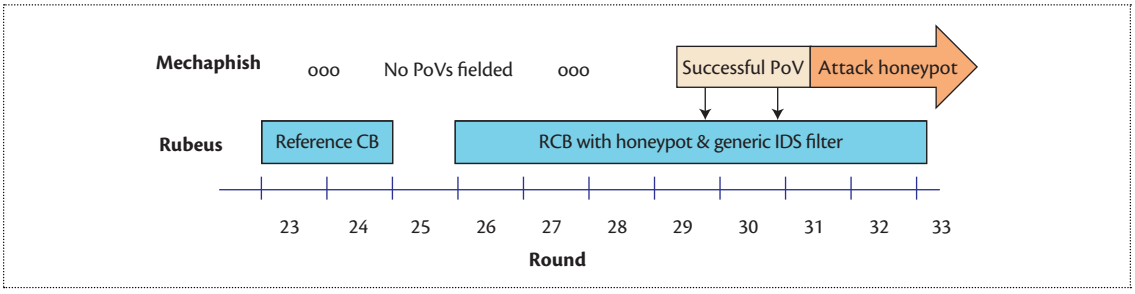


Figure 2. Mechaphish PoVs against Rubeus for service KPRCA_00065.

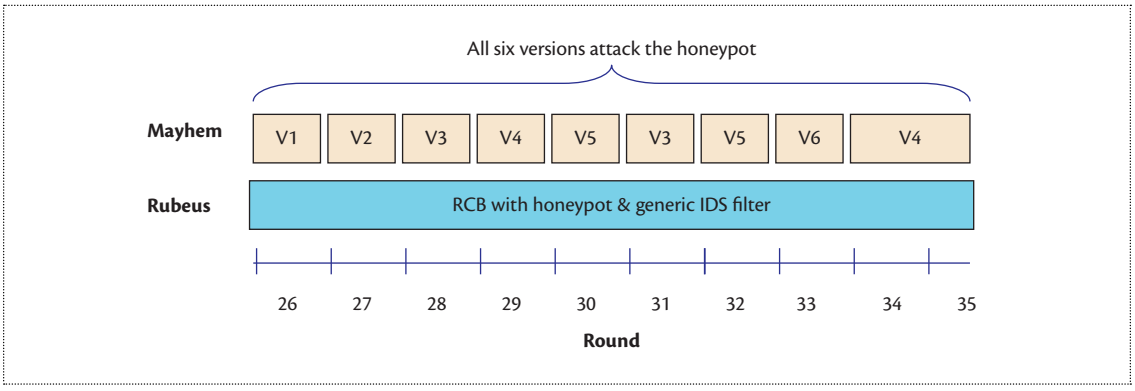


Figure 3. Mayhem PoVs against Rubeus for service KPRCA_00065.

the RCB had the CRS not been distracted by the honeypot.

The game infrastructure provided feedback to each CRS, indicating the success of their PoVs. The Galactica CRS could have used that feedback to determine that attacking the honeypot did not lead to successful PoVs. The fact that the CRS attacked the honeypot in five different services, 30 different times, suggests the CRS did not advantageously respond to the feedback. Other than the successful PoVs that they abandoned, Galactica successfully exploited honeypotted RCBs for CROMU_00065, after scoring against a non-honeypotted RCB for that service (which Rubeus churned five times).

Mechaphish

As illustrated in Figure 2, Mechaphish abandoned a PoV against KPRCA_00065 that succeeded in two consecutive rounds in favor of an attack on the honeypot for the next five rounds. While other teams repeatedly deployed PoVs that the game infrastructure indicated were impotent, only Mechaphish and Galactica did this after obtaining feedback reflecting the success of previous PoVs against the same RCBs. Mechaphish repeatedly attacked honeypots in three services, and they scored against five other Rubeus services, four of which were honeypotted.

Mayhem (aka Pooh Bear)

Mayhem attacked honeypots in seven services dozens of times. The CFE champion never scored against a honeypotted RCB. The one Rubeus RCB that Mayhem scored against was a service that was never honeypotted. While they repeatedly deployed the same impotent PoVs against honeypots, they also sometimes responded to game infrastructure feedback by frequently deploying several different PoVs against the honeypot for the same service. For example, Figure 3 shows that during 10 rounds of KPRCA_00065, Mayhem deployed six different PoVs, each targeting the honeypot. During those same 10 rounds, Xandra scored, and during two of those rounds, Mechaphish scored before being lured by the honeypot (as previously shown in Figure 2).

Mayhem had working PoVs for several of the services for which it targeted honeypots. Given that Rubeus rarely mitigated the vulnerabilities in RCBs, it is likely that Rubeus would have been scored on by Mayhem were it not for the honeypot.

Crspy

Crspy attacked honeypots in three RCBs of two services and repeated the same PoV only once. Crspy never scored on Rubeus. The data for Crspy raises the question of why their CRS attacked the honeypot only a few times. Why would a CRS exhibit a predilection for a honeypot in only a few of the many RCBs in which it appears?

Jima

This CRS did not appear to deliberately attack the honeypot. Jima did throw a generic prebuilt PoV that would have scored on the honeypot in an RCB in two rounds. The fact that a PoV is prebuilt does not preclude the CRS from determining that the PoV may be effective against an RCB (or at least appear to be). However, Jima threw this particular PoV against most services in most rounds, leading us to conclude that their CRS did not specifically target the honeypot.

Methodology

The CGC infrastructure logs[5] identify which RCBs were fielded by Rubeus and which competitor PoVs were successful against those RCBs. Each Rubeus RCB was analyzed with IDA Pro to identify those containing the honeypot, which was not obfuscated and was the first function invoked by the binary.

Identification of which teams targeted the honeypot with a PoV could not be directly derived from game logs, and is less precise. These occurrences were discovered using an analyst replay tool developed as part of the system used to vet competitor software deployed in CFE.[6] This analysis tool was reconfigured to emulate the CGC infrastructure as it would have looked without employing the CGC DECREE hypervisor, thereby causing the honeypot code branch to be taken. In this configuration, the analysis tool replayed each PoV thrown against each Rubeus RCB. However, only one session for each PoV/RCB pair was replayed, using the session seeds from the first throw of one round in which the pair occurred in the game. Each PoV that succeeded against the honeypot in these trials was assumed to have succeeded in other sessions. This methodology assumes no reliance on randomness in the subject sessions. Analysis of a sample of the subject PoVs suggests they were deterministic (excluding the Jima prebuilt generic PoVs that happened to exploit the honeypot).

The honeypot successfully lured several competitors, leading some to abandon working PoVs. On the other hand, use of the honeypot significantly affected Rubeus's availability scores due to fielding RCBs whose honeypot logic led to immediate crashes and rounds lost due to the resulting churning of RCBs. The honeypot does not appear to have materially affected the outcome of the competition. Its precise effect on team scores is difficult to determine because we do not know whether CRSs would have deployed successful PoVs in the absence of the honeypot or if Rubeus would have revised its defenses if successfully attacked.

The Rubeus CRS did not appear to include functions to determine if an RCB would fail service polls. Competitors were provided a network tap of all traffic directed at their system. The CGC competition developers expected teams would use this traffic to reproduce service polls to determine if candidate replacement binaries broke service functionality. Had Deep Red implemented this, their CRS might have not deployed so many RCBs (and IDS filters) that failed all service polls, and thus they may have avoided much of the damage due to the flawed honeypot execution divergence logic.

Honeypots in the form of deliberate software vulnerabilities become effective when those vulnerabilities are visible to potential attackers. In the context of capture-the-flag (CTF) competitions, this would require some form of consensus evaluation, which is not often part of CTFs. But for those CTFs that do make team patches available to other teams for review, honeypots could be effective. Regarding real-world network defense, adversaries often have access to the binary code for services run by their intended victims. However, publishing deliberately vulnerable software with the goal of fooling an attacker can also potentially lure an unwitting third party into relying on that same software. That third party may not be aware of the vulnerability or the means of controlling its exploitation. ■

Acknowledgments

This work was sponsored by the Defense Advanced Research Projects Agency. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

References

1. "Cyber Grand Shellphish," Team Shellphish, 2017; phrack.org/papers/cyber_grand_shellphish.html.
2. *Intel 64 and IA-32 Architectures Software Developer Manual*, vol. 2, Instruction Set Reference, Intel, 2016.
3. "Cyber Grand Challenge: Rules," DARPA, 18 Nov. 2014; archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_Rules_18_Nov_14_Version_3.pdf.
4. *Intel 64 and IA-32 Architectures Software Developer Manual*, vol. 3a, System Programming Guide, Part 1, Intel, 2016.
5. "Cyber Grand Challenge: CGC Corpus," Lunge Technology, 2017; www.lungetech.com/cgc-corpus/cfe.
6. T. Vidas et al., "Designing and Executing the World's First All-Computer Hacking Competition: A Panel with the Development Team," Schmoocon, 2017; https://archive.org/details/ShmooCon2017.

Michael Thompson is a research associate at the Naval Postgraduate School (NPS) in Monterey, California. His research interests include network security simulation-based educational games, high-assurance multilevel security, and software vulnerability analysis tools. Contact at mfthomps@nps.edu.

Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge

Anh Nguyen-Tuong | University of Virginia
David Melski | GrammaTech
Jack W. Davidson, Michele Co, William Hawkins, and Jason D. Hiser | University of Virginia
Derek Morris | Microsoft
Ducson Nguyen and Eric Rizzi | GrammaTech

For the Cyber Grand Challenge, competitors built autonomous systems capable of playing in a capture-the-flag hacking competition. In this article, we describe the high-level strategies applied by our system, Xandra, their realization in Xandra's architecture, the synergistic interplay between offense and defense, and lessons learned via post-mortem analysis.

The challenge in DARPA's Cyber Grand Challenge (CGC) was to build an autonomous system capable of playing in a capture-the-flag hacking competition. The final event pitted the systems from seven finalists against each other, with each system attempting to defend its own network services while proving vulnerabilities ("capturing flags") in other systems' defended services.

The competition was organized around a collection of challenge sets (CSs), each consisting of one or more challenge binaries (CBs) that implement a network service. Each team in CGC developed a cyber reasoning system (CRS) that interacted with the competition framework (CF) to retrieve CBs; analyze the CBs to identify vulnerabilities; upload hardened replacement challenge binaries (RCBs); upload rule sets for a network intrusion defense system (IDS) to monitor and/or modify network traffic; upload proofs of vulnerability (PoVs) against competitors' services; and obtain feedback about fielded CBs, IDS rules, PoVs, and scores.

Each CRS was scored in each round of the competition for each CS live in that round:

- *Availability* measured the preservation of a CS's required functionality and overhead. It took values in the range 0–1. There were some allowances for overhead (20 percent file size, 5 percent memory, 5 percent performance), but steep penalties were imposed for exceeding that allowance or breaking functionality.
- *Security* measured the ability to block competitors' attempted PoVs. This score was 1 if any competitor succeeded in landing a PoV against a CS, and 2 otherwise.
- *Evaluation* measured the ability to land PoVs against competitors' services. It took values in the range 1–2 and was proportional to the number of competitors that were successfully PoV'd.

The total score for a CS in a round was computed as: availability × security × evaluation × 100. At the end

 Copublished by the IEEE Computer and Reliability Societies

of the game, each team was ranked based on the total score aggregated across all rounds and all CSs.

There were two primary mechanisms to defend a service: patching the CBs in a CS or installing IDS rules. Note that defending using either mechanism rendered the CS unavailable for one round, during which the score for the CS was 0. During this down round, replacement binaries or replacement IDS rules were made available to competitors for analysis.

There were two ways to prove a vulnerability: a Type 1 PoV represented a control flow hijacking attack and required demonstrable control of the instruction pointer and one other register. A Type 2 PoV represented an information leakage attack and required leaking and reporting four consecutive bytes from a designated memory page.

Scoring Tradeoffs

The scoring system precluded various strategies and favored others (see the DARPA Cyber Grand Challenge Competitor Portal for details: http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite).

For example, we initially considered a moving target defense strategy in which Xandra would replace binaries with a new, diversified variant every *n*th round, thereby forcing competitors to reanalyze our replacement binaries. The scoring penalty for fielding replacements made this strategy unworkable.

Getting a full score for security required defending against all competitors, making defenses that cover an entire attack class attractive. However, the strict performance and memory target envelopes of 5 percent favored strategies that were precise and that perturbed CBs only as needed to prevent a successful PoV.

On the offensive side, successful PoVs acted as bonus multipliers because there were no penalties associated with throwing PoVs against competitors. The initial rounds when a new CS was introduced by the competition framework was the only time a CS was guaranteed to be identical for all competitors. Thus the scoring rules provided strong incentives for finding PoVs quickly, before competitors could field defenses.

Xandra Strategy

The main elements of our strategy were:

- *Offense.* Use a combination of fuzzing and symbolic execution to find crashing inputs, and convert crashing inputs into PoVs.
- *Defense.* Defend at most once, and only when there is an indication that a CS might be vulnerable to a PoV: use control flow integrity (CFI) techniques to prevent Type 1 PoVs, use network filters to prevent Type 2 PoVs, and use point patching judiciously.
- *Safety.* Roll back any replaced binary with the original version when something goes wrong, for instance, if availability falls below a set threshold.

Our use of broad defenses provided the rationale for defending at most once (if a defense truly covers an attack class, there is no need to redefend). This strategic decision turned out to be crucial and well-adapted to the way the final game play unfolded.

However, broad defenses are generally costly in terms of memory and performance overheads. A primary challenge leading up to the final event was to develop lightweight defenses that retained the strong security property of covering an attack class while meeting the overhead constraints imposed by CGC.

Architecture

Xandra implements a virtual capture-the-flag team, with individual components that correspond to different roles. Figure 1 illustrates Xandra's architecture and workflow. At the core of the architecture is the GameMaster (GM), which is responsible for maintaining situational awareness; managing the work of other components; and submitting replacement binaries, IDS rules, and PoVs.

Upon detection of a new challenge set, the GM forwards the associated CBs to both offensive and defensive components simultaneously. In cases when a competitor uploads a new RCB, the RCB is forwarded only to the offensive component.

The offensive component is divided into two primary subcomponents: a fuzzing pod whose goal is to find inputs that result in a crash and a PoV generator that turns crashing inputs into PoVs.

The goal of the defensive component, Helix, is to generate functionally equivalent, protected, and efficient binaries. The default security policies are to armor binaries with CFI techniques to protect against Type 1 PoVs and to deploy a fixed set of IDS rules to handle Type 2 PoVs.

When a crashing input is found, the dynamic analysis engine (Daffy) generates point-patching policies and invokes Helix to reprotect binaries with the point patch applied.

Xandra uses the OpenStack cloud management infrastructure (https://www.openstack.org) to provision computing resources and to isolate components from one another. Xandra employs a bag-of-tasks model for both offensive and defensive components, using the Beanstalk work queue system (http://kr.github.io/beanstalkd). The advantages of this architecture are well-known: self–load balancing of tasks and natural fault-tolerance capabilities.

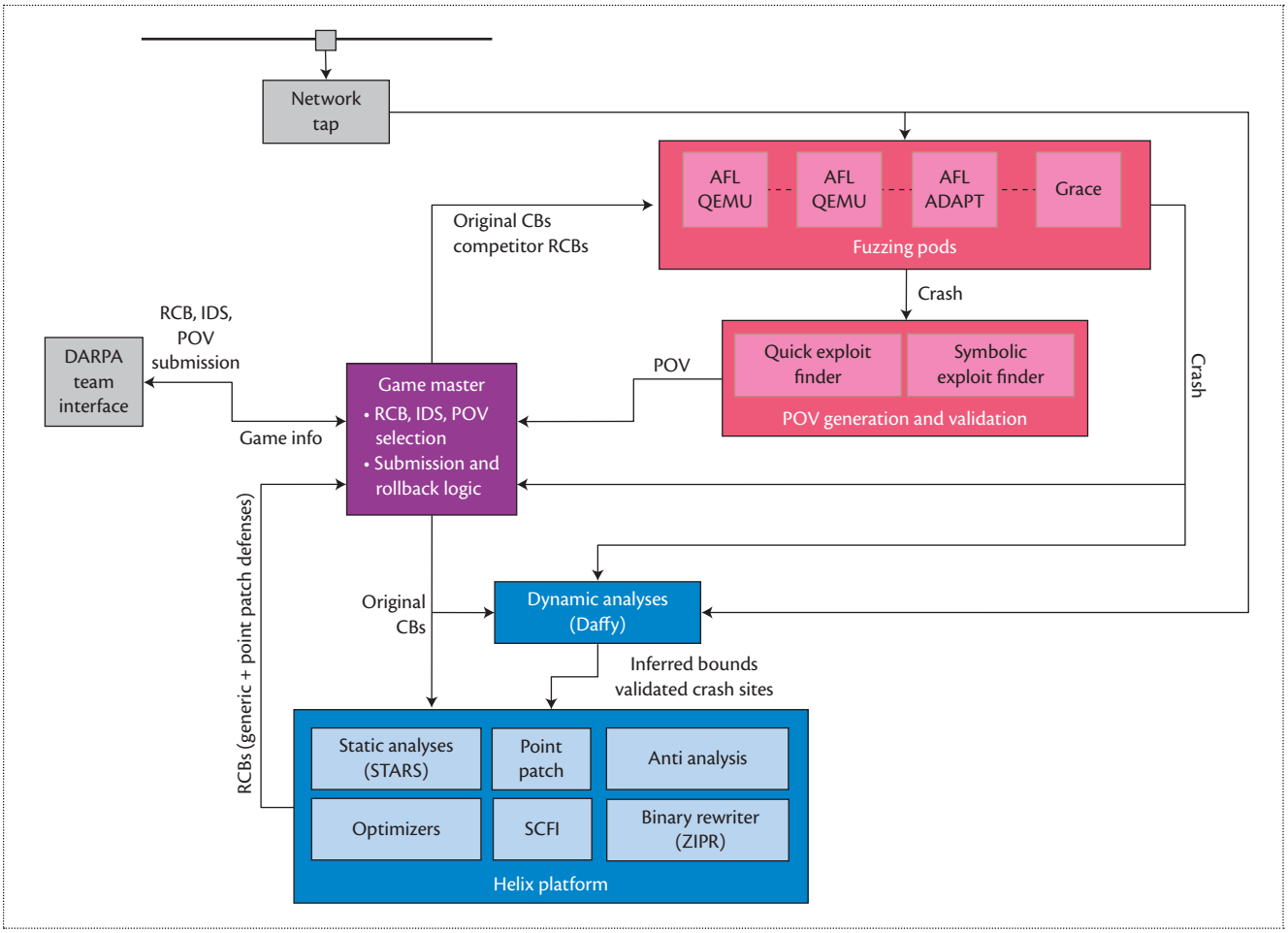


Figure 1. Xandra high-level architecture. The GameMaster is responsible for interacting with the competition framework (via the Team Interface) to carry out the game strategy by coordinating activities between offensive components (red) and defensive components (blue). Both offensive and defensive components had access to anonymized network traffic provided by the competition framework.

GameMaster

The GM manages the strategy described earlier and periodically polls the Team Interface for new information about the game state, for instance, current round number, new challenge binaries, replacement binaries and IDS rules fielded by competitors, and feedback on previously submitted binaries.

When Xandra successfully generates a new PoV, the GM immediately submits the PoV to the CF, provided that the reliability estimate for the PoV is higher than previous PoVs.

However, when Xandra generates a new protected binary, the GM waits to submit the binary until either of the following conditions is met:

- Xandra finds a crashing input.
- Feedback from the CF indicates that the original CS fielded has exited abnormally—that is, a competitor found a crashing input.

In other words, Xandra uses crashes as a proxy for exploitability.

When the GM decides to submit a replacement binary, it also submits a network filtering rule in the same round, provided the rule is unlikely to break functionality.

The GM performs a rollback if the CF reports broken functionality or if performance/memory overhead is higher than a somewhat arbitrary threshold of 30 percent.

Offense

Xandra uses both fuzzing, which (semi-) randomly generates test inputs, and symbolic execution, which generates and solves logical constraints to find inputs that lead to desired execution traces. Xandra's component for vulnerability discovery is called a *fuzzing pod*, even though it performs both fuzzing and symbolic execution. We provisioned sufficient fuzzing pods to analyze

30 CSs and RCBs from six competitors concurrently. Each fuzzing pod runs eight instances of the fuzzer and one instance of the symbolic execution engine. The fuzzing pod leverages captured inputs provided by the CF as seeds for the fuzzers, which can dramatically improve their effectiveness. Nondeterminism in a protocol, such as the use of nonces, can limit the utility of captured inputs as fuzzing seeds. Xandra uses a technique called *(de)noncification* to transform captured inputs into better seeds.

American Fuzzy Lop and extensions. Xandra's fuzzer is based on American Fuzzy Lop (AFL; http://lcamtuf.coredump.cx/afl), an industrial-grade fuzzer. AFL uses profiling to fingerprint the behavior exhibited by the subject program when run on an input. Inputs that lead to distinctive fingerprints (profiles) are given preference for further mutation and testing. AFL's performance is directly tied to how quickly it can generate profiles.

Xandra's fuzzing pods run a mixture of AFL instances in different configurations. Some instances use emulation (based on QEMU) to gather profiles, and some use binary instrumentation. The latter is typically 3x more efficient, but either technique can be foiled by antitamper measures. Xandra dynamically balanced the use of the two approaches on each CB based on observed effectiveness.

Fuzzing can be ineffective at triggering behavior that requires very specific inputs (for instance, "easter eggs," or hidden functionality).

We optimized AFL's profiling for use in CGC as follows:

- *Remove nondeterminism.* Nondeterminsm can lead to "noise" in AFL's profiles, which may negatively affect selection of inputs for mutation. Fortunately, the only source of nondeterminism in CBs was the random system call and the "secret page." We initialized these with fixed values during vulnerability discovery.
- *Skip transmit system calls that output to stdout or stderr.* AFL considers only the edge profile generated by executing on an input and ignores output by piping `stdout` and `stderr` to `/dev/null`. However, the program still incurs the overhead of placing the system call. We removed that overhead.
- *Skip unnecessary receive system calls.* A receive requesting zero bytes is treated as a no-op, and the system call is skipped.
- *Input delay heuristics.* When running a subject CB, AFL pipes the input to the CB's `stdin` file descriptor. Doing so means that the entirety of the input is immediately available to the CB. However, many CBs use a protocol that assumes there will be intermittent pauses in the delivery of the input. We implemented simple heuristics that model delays in the availability of input on `stdin` when the CB calls `fdwait`.
- *Skip sleep calls.* We developed simple heuristics to dramatically reduce sleep delays or skip sleeps entirely.
- *Late forking.* On Linux, launching a program requires a call to `fork()` to create a child process followed by a call to `exec()` to replace the child's image with the desired program. AFL modifies the subject program so that the `exec()` only needs to be done once to create an instance of the program that will act as a fork server. Thereafter, AFL alerts the server via a pipe to trigger forking of a child process that will continue execution on a new input from AFL, without the need for another call to `exec()`. By default, AFL places the fork server so it will be executed (and take control) before the subject program has done any computation. To avoid repeated and unnecessary execution of a CB's initialization code, we moved the fork server initialization to the first call to `receive()`.

Given the nondeterminism of fuzzing and the large number of CBs, we did not have sufficient resources for precise evaluation of the fuzzing pods. In a small set of timed experiments on 80 CBs from the CGC Qualifying Event, our AFL optimizations increased the number CBs crashed from 14 to 35.

Grace2 symbolic execution engine. AFL is effective at finding crashing inputs for the CSs in CGC, especially given seed inputs that demonstrate how to exercise the CS's functionality. However, fuzzing can be ineffective at triggering behavior that requires very specific inputs (for instance, "easter eggs," or hidden functionality). For example, fuzzing is unlikely to guess the one integer that would satisfy the condition (`user_int == MAGIC_CONSTANT`). In contrast, the constraint solvers used by symbolic execution have no trouble identifying an input that satisfies this condition.

Xandra augments AFL with symbolic execution to address this potential shortcoming with fuzzing. We built our symbolic execution engine, Grace2, to augment AFL. Its design is very similar to Driller,[1] the symbolic engine developed by Shellphish. In contrast to Driller, Grace2 immediately starts selecting inputs generated by AFL and attempts to mutate them to generate

inputs that will drive the program toward new, interesting states. Like Driller, Grace2 uses the edge profiles favored by AFL to determine whether or not an input is interesting.

PoV generation. Xandra employs redundancy for several of its capabilities. It uses two approaches to PoV generation: *Quick Exploit Finder* (QEF) and *Symbolic Exploit Finder* (SEF).

The first, QEF, uses inductive experimentation to determine if an input can be converted to a Type 1 or Type 2 exploit. It is simple, fast, and capable of working on any CS, including those that are multiprocess or tamper-proofed. However, it will succeed only for trivially exploitable vulnerabilities.

For Type 1 exploits, QEF runs the CS on a crashing input and looks for correlation between the register values at the crash site and the file contents. Wherever it finds a match, it mutates the appropriate file location and observes the effect (if any) on the register values if the program still crashes on the mutated input. If QEF found a way to control the IP register but not another register, it would attempt to find a portion of memory copied from the input and craft a code injection attack.

The search for Type 2 exploits begins during vulnerability discovery. The fuzzing pods monitor the CS output for the values we used to initialize the secret page and for simple transformations of those values. If a match is found, we experiment with modifications to the secret page to determine if we have a reliable Type 2 exploit.

QEF had the advantages of simplicity and efficiency. However, even minor transformations on the input could foil its ability to generate an exploit. To compensate, we developed the Symbolic Exploit Finder. SEF uses Grace2 to symbolically execute crashing inputs up to the crash site. At the site of the crash, it attempts to discover how much control it has over the registers and memory values involved in the crash, which helps it determine masks to use for Type 1 PoVs. To determine control of each register, SEF uses the symbolic expression for the register value. SEF uses a series of solver calls to identify the "fixed bits" that are the same in any execution. The remaining bits may still be constrained, though they differ in some executions. SEF performs a bounded search over candidate masks evaluating each candidate against randomly generated challenges, as the final PoV has to do.

If SEF determines it has sufficient control, it generates a set of constraints on the inputs that must be true for the input to meet the challenges produced by the CF. These constraints are then inserted into the SEF template PoV. The template PoV contains a copy of the Yices 2 SMT solver. Each template runs the script necessary for communicating with the CF during execution of a Type 1 PoV. At the appropriate place, it solves the constraints generated by SEF and feeds them to the subject binary. If a straightforward Type 1 exploit is not possible, SEF will also attempt an injection of a simple 12-byte program to set register values and a write-what-where attack to corrupt a pointer value.

During the competition, QEF was sufficient for the PoVs Xandra found.

PCAP Maximizer

Out of the box, AFL often struggles to achieve sufficient coverage on a given binary. One of the best ways to increase AFL's efficacy is to seed it with sample inputs that exercise various parts of the program. The CF provided this information in the form of PCAP files representing sample interactions with the programs. At a high level, our CRS captured this information and fed it into the each of the AFL instances running in the fuzzing pods. Just feeding these PCAP files naively into the various instances of AFL running under the fuzzing pod would have, in many cases, been suboptimal. To maximize the efficacy of these sample inputs, we did three things.

> **One of the best ways to increase AFL's efficacy is to seed it with sample inputs that exercise various parts of the program.**

Reduce PCAP file size. The sample interactions coming in from the CF were often too long; AFL does best with short inputs that can be run thousands of times per second. To shorten the inputs without losing unnecessary information, we reduced each PCAP file to the point that it would run through AFL within 25 ms. Like many parameters in the fuzzing pod, 25 ms was chosen based on AFL documentation and minimal experimentation.

Use AFL to iteratively noncify the program. The second complexity we had to address was the fact that each PCAP file provided by the CF was the output of a binary run under an unknown random seed. The use of random seeds meant that if there were any special tokens (nonces) in the program, playing back the same input on our own system would result in a different path being followed. For example, consider the program `KPRCA_00001`. This program gives the user a special key, based on some unknown random seed, that

must be input back into the program to access the main functionality of the binary. The input from this program looks something like this:

```
USER: Hello
PROGRAM: Your user ID is 648183
USER: 648183 says get /home/user
PROGRAM: Retrieving contents of /home/
user
...
```

Were we to naively rerun a sample input provided by the CF, we would almost certainly (because our random seed would be different) fail the playback test and not be able to exercise the same paths as the sample input. Instead, we would have gotten something like the following:

> **In contrast to most mutation strategies, noncification may use portions of a program's output (responses) to mutate the input.**

```
USER: Hello
PROGRAM: Your user ID is 715222
USER: 648183 says get /home/user
PROGRAM: Sorry, incorrect user ID,
goodbye!
```

To handle this problem, we leveraged AFL's profiles to "noncify" the input. That is, we looked for areas where the input and the output of the sample PCAP file matched. We considered these candidate nonces. We then tried various snippets of the sample PCAP input as nonces, each time running it through AFL to see whether the created variant resulted in a new path. We continued to do this in an iterative fashion, exploring each PCAP file (and its possible nonces) as deeply as possible.

Using AFL profiling, we could test multiple input variants very quickly, allowing us to handle cases where nonces were of an unknown length or ended with an unknown delimiter. In effect, we consider "noncification" to be a different mutation strategy that augments the mutation strategies used by AFL. In contrast to most mutation strategies, noncification may use portions of a program's output (responses) to mutate the input.

Focus on new interesting paths. We had to consider the large number of PCAP files that would be coming in from the CF. Given our understanding of the game specifications, we expected there to be hundreds of samples per minute. Given the number of inputs, it was likely that many of them would trace paths already explored by AFL. Therefore, we created one noncifier job specifically to handle the incoming PCAP file. After noncification, the noncified input would be distributed for further analysis by fuzzing and symbolic execution.

Defense

The Helix platform provided Xandra with the ability to transform binaries arbitrarily and efficiently. A key component of Helix is Zipr, an efficient static rewriter that transforms binary programs and libraries without access to their source code.[2] Zipr is compiler agnostic and applicable to both dynamically and statically linked programs and shared and static libraries. We have used Zipr to successfully rewrite such large code bases as libc, OpenJDK's libjvm, and the Apache webserver and its modules, and felt confident in its applicability to CGC binaries. The output of Zipr is a compact, transformed program or library with all the functionality of the original (unless that behavior is explicitly modified) and new functionality added through the application of composable user-specified transformations.

The primary transformations used by Xandra are block-level instruction layout randomization, selective CFI, point-patching techniques, various binary optimization techniques, and anti-analysis techniques. In addition to rewriting binaries, we also used network filters—all addressed below.

Block-level instruction location randomization (BILR). BILR is a diversity technique that randomizes the location of instructions by relocating code at the block level of granularity. Our original intent was to use BILR as a moving target defense to force competing teams to reanalyze and reattack our RCBs. However, the one-round penalty for submitting RCBs made this strategy prohibitively expensive. Instead, we used BILR as a defense-in-depth technique, coupled with selective control flow integrity (SCFI).

Selective control flow integrity. Control flow integrity techniques seek to ensure that control flow transfers adhere to a control flow graph specification.[3] To protect against Type 1 PoVs, we used a selective form of SCFI with the following characteristics:

- SCFI extracts the control flow graph directly from the binary.

- SCFI is coarse grained. All indirect control flow transfers—targets of indirect jumps, calls, and returns—belong to the same target class.
- SCFI is selective. We analyze binaries and look for safe indirect transfers as these need not be instrumented with CFI checks.

We classify functions as safe when we can prove that the return address for a given function cannot be overwritten. The proof enables us to skip the CFI instrumentation for returns from safe functions. We also forgo CFI instrumentations for switch tables. Omitting unnecessary CFI checks was key to obtaining good performance.

Because of the tight 5 percent constraint on memory and performance, we opted for a one-byte executable nonce, potentially at the risk of a larger attack surface.

Despite well-known weaknesses with coarse-grained CFI in general,[4] we felt that SCFI would be effective within the context of CGC—that is, it could be implemented efficiently and would raise the bar for competitors significantly.

Sample SCFI instrumentation is shown below:

```
(1) ...  ; at call to foo():
(2) call foo
(3) nop ; 1-byte executable nonce 0x90
(4)  ...    ; at return from foo():
(5) and [esp], 0x7FFFFFFF ; clamp
(6) mov ecx, DWORD [esp] ; assume ecx
    free
(7) cmp BYTE [ecx], 0x90 ; verify nonce
(8) jne _terminate
(9) ret
```

In this example, our instrumentation inserts a one-byte `nop` executable nonce after the call instruction (line 3). In function `foo()`, the return site is augmented to check for the presence of the executable nonce (lines 6–8). If an attacker is able to control the return address, the check for the executable nonce should fail. Because the stack is executable, we had to handle the case where an adversary may guess our executable nonce and insert malicious payload on the stack. To prevent control flow transfer to the stack, our CFI instrumentation clamps the return address (line 5) so transfers to code on the stack are not possible. We considered using a larger executable nonce, for example, two or more bytes, to increase the difficulty of finding return-oriented programming (ROP) gadgets. However, because of the tight 5 percent constraint on memory and performance, we opted for a one-byte executable nonce, potentially at the risk of a larger attack surface.

Daffy and point patching. Daffy is the name of Xandra's dynamic analysis engine. It is implemented on top of the Pin dynamic binary translator.[5] Daffy serves two primary purposes: bounds inference for stack arrays and generation of rules for faulting instructions. Daffy's approach was inspired by the dynamic reverse-engineering tool, Howard.[6] Daffy built up a database of memory access patterns based on traces of (presumed) benign, non-crashing inputs and contrasted them with accesses that caused a CB to crash. Based on the difference, Daffy inferred (possibly unsoundly) a memory access invariant. For a given CB, the output of Daffy is a set of faulting instructions along with policy rules to be applied (that is, bounds check or clamp).

In addition to broad-based defenses, Xandra also applies point patches for stack-based arrays. In cases where Daffy is able to infer array bounds, a bounds check is inserted. In cases where Daffy cannot infer bounds for the faulting instruction, a clamping technique is applied to instructions that read memory and load the value into a register. For example,

```
mov ecx, dword [eax]
```

when clamped, becomes

```
lea ecx, [eax]
and ecx, 0xbfffffff
mov ecx, dword [ecx]
```

The clamp value modifies only one bit and ensures that the memory read cannot originate from the flag page. The effect of clamping is to turn a Type 2 PoV attempt into a program crash.

Binary optimizations. To amortize the cost of our primary protections, we incorporated several optimization phases into the Helix tool chain. These included peephole optimization, register allocation, and CRCX eliding.

The first two techniques are standard compiler optimizations that were included because we noticed that many of the CBs used in the qualifying event and during the trial sparring sessions leading to the final event were not optimized (on some CBs, performance gains were substantial, up to 15–30 percent).

The third, *CRCX eliding*, was a CGC-specific technique. During testing, we noticed that DARPA added

an integrity check to all CBs, which we will call CRCX here. This code would run before the actual application code. Because this hash was immediately discarded after generation, this code had no real effect other than a performance and memory hit. To remove this code, we constructed a small tool that located points in the CB code where the CB had returned to the initial machine state but at a later execution point. The tool would update the entry point of the CB to be the later execution point. Subsequently, any CRCX code would be elided from the final binary as Zipr removes unreachable code blocks automatically.

Anti-analysis techniques. We developed three anti-emulation techniques, two of which targeted QEMU explicitly. To mitigate risks and overhead, we deployed only the two static techniques.

- *Zero page.* This technique exploits a bug wherein QEMU rejects binaries with headers that have zero for the size, mapped size, offset, and address.
- *String table.* This technique targets *objdump*, *gdb*, and similar programs. The key idea is that the kernel does not care about the section headers in the binary, so corrupting data related to them should only break analysis tools. We changed the listed number of sections to be 0, and the number referencing the index of the string table in the sections to be very large. These changes cause rejections or crashes in a number of standard analysis programs.
- *Floating-point divergence.* Our third technique was dynamic and relied on a floating-point divergence between the CGC operating system and QEMU for the `sinf` instruction. This divergence happened only for one specific value out of 2^{32} possible values.

We considered bootstrapping this last technique to crash the program, deceive symbolic engines into exploring decoy paths, induce infinite loops, and attempt to escape QEMU to interfere with competitors' CRSs. We decided against this course of action, because we already had a static, less expensive anti-QEMU technique and, most important, because we were not sure how far we could "push the limit" and stay within CGC rules. However, we could not rely on competitors to behave similarly. To mitigate the risk to Xandra, we took great care to isolate competitors' RCBs from critical components of Xandra.

Network defenses. Xandra has a single, fixed network filter rule set that it selectively deploys when it decides that defense is required and the rule set has a low probability of breaking functionality. Xandra's rule set is designed to block injections of addresses that reference the CGC secret page (in the range `0x4347C000` – `0x4347CFFF`). Xandra's rule set consists of 16 blocking rules, one for each possible value for the three high-order bytes on a secret-page address.

Xandra's rule set is prone to both false negatives and false positives. A false negative may occur if the injected address is encoded in any way. For example, a false positive would occur if the address is input as a decimal string and then converted by the targeted application. Despite this possibility, our hope was that the rules would block some attacks.

A false positive may occur if a secret-page address occurs (for instance, randomly) as part of benign input. To avoid false positives, Xandra checks for occurrences of secret-page addresses in the client-side network traffic that is captured in the round immediately after a

Table 1. CGC Final Event rankings.

CRS	Security	Evaluation	Availability
1. Mayhem	#6	#6	#1
2. Xandra	#1	#4	#2
3. Mechaphish	#2	#1	#5
4. Rubeus	#3	#3	#4
5. Galactica	#4	#2	#6
6. Jima	#7	#7	#3
7. Crspy	#5	#5	#7

Table 2. Net scoring defensive gains.*

CRS	Never PoV'd	PoV'd	Defensive gains
1. Mayhem	(477)	8,849	8,372
2. Xandra	(13,441)	15,071	1,630
3. Mechaphish	(25,308)	13,162	(12,146)
4. Rubeus	(10,901)	473	(10,429)
5. Galactica	(25,385)	8,188	(17,197)
6. Jima	(10,903)	244	(10,659)
7. Crspy	(27,971)	3,280	(24,690)

**Except for Mayhem and Xandra, cost of defenses on never-PoV'd CSs outweighed benefits of defending against PoV'd CSs.*

challenge set is released, before competitors have an opportunity to field PoVs. If Xandra detects any (apparent) secret-page addresses in this presumed benign input, the rule set is not deployed.

Postmortem Analysis

Table 1 provides the final rankings for each CRS. It is intriguing to note that Mayhem finished first despite being ranked #6 for both security and evaluation. Mayhem suffered a catastrophic failure that rendered it inoperable for one-third to half the event. However, by that time, Mayhem had built up a sufficient lead that it never relinquished.

Availability turned out to be the most important metric as only 20 out of 82 CSs (24 percent) were PoV'd. For the others, the optimal strategy would have been to leave the original CS alone and not attempt any defensive actions, as even a perfect replacement binary or IDS rule incurs a one-round availability penalty. While Xandra's high-level strategy to field an RCB at most once turned out to be a good fit, a key improvement for future competitions would be to refine Xandra's use of crashing as a proxy for exploitability and take into account the difficulty of turning crashes into PoVs.

Table 2 shows the total defensive gains realized by each team. Defensive gains or losses were computed by comparing availability and security scores against a no-defense policy. Given the ratio of PoV'd CSs, every team except Mayhem and Xandra could have improved their scores by employing a no-defense strategy!

Resiliency

One notable result during the competition was the importance of avoiding denial of service (DoS) through CRS overloads. Both Rubeus and Mechaphish had rounds where the availability scores plummeted for all (or most) CSs active in those rounds. This result implies they suffered essentially a DoS on the machine running their services during those rounds. Mayhem also seemed to suffer from a self-DoS from a bug that rendered their system inactive for a third to a half of the competition.

Xandra was carefully designed to minimize the chances of DoS. We were conservative in our application of defenses and our use of resources (Xandra drew the least power of any of the competitor systems). Possibly we did not maximally leverage the resources of our cluster, but we also avoided catastrophic failures.

Offense

Xandra succeeded in finding crashing inputs for many of the CSs and RCBs fielded by competitors. We suspect that some of the crashes, for instance, RCBs from Rubeus, were designed to derail analysis and were not actually exploitable. Considering just the original CSs, Xandra found crashing inputs from 38 of the 82 CSs, including 14 of the CSs that were proven vulnerable during the CGC Final Event (CFE).

Xandra's biggest weakness seemed to be converting crashing inputs into exploits. During the CFE, Xandra produced just four PoVs. In experiments after the CFE, Xandra regularly produced a PoV for a fifth CS, which demonstrates the nondeterminism of our fuzzing pods. After a bug fix, it was able to generate a PoV for a sixth CS. In addition, there were two more crashing inputs that gave control over the instruction pointer but not a general-purpose register. With a little more work on generating ROP exploits, Xandra would probably be able to generate PoVs for those as well.

Xandra made extremely good use of the vulnerabilities it did discover. Even though Xandra was sixth in terms of number of CSs proven vulnerable, it was fourth in its evaluation score. Many competitors seemed to focus on sophisticated vulnerability and exploit discovery. This strategy allowed them to find more exploits, but slowly. In contrast, QEF was very quick and, when it created an exploit, allowed the exploit to be thrown in more rounds.

Many competitors seemed to focus on sophisticated vulnerability and exploit discovery. This strategy allowed them to find more exploits, but slowly.

Defense

When deployed, Xandra's defenses were extremely effective during the competition. We found only two examples, one Type 1 and one Type 2, where a competitor's CRS managed to bypass Xandra's defenses.

Patching and IDS rules. Xandra replaced 42 of the CSs. On 26 of those, we applied only generic defensive policies. On 16, we also applied point patches.

Most RCBs had high availability (six were within 5 percent and 18 within 10 percent of the original) though three RCBs incurred severe penalties (more than 35 percent). Xandra reverted eight of the 42 RCBs it replaced (seven due to broken functionality, one due to exceeding our memory threshold). It also failed to revert two RCBs that should have been rolled back due to low availability scores but were not, due to a bug in the GameMaster logic.

Of the seven rollbacks caused by broken functionality, one was due to the rewriting process, another

due to control flow integrity, and yet another due to a CS performing an integrity check over its text and data segments. Four rollbacks were due to point patches that Xandra applied. An assumption behind the clamping policy is that it would be safe to clamp a faulting instruction. However, we should have realized that the faulting instruction could also access the flag page under benign inputs. A safer policy would have been to detect potential false positives, similar to what was done for the network filter rules, which we discussed earlier. Even then, our IDS rules also exhibited false positives.

We intended to add a module to Xandra that would use outputs from the PCAP maximizer to validate RCBs prior to deployment, but due to time and resource constraints, we settled for a much simpler validation process that consisted of testing the validity of RCBs using simple fixed inputs. This module would have dramatically increased Xandra's availability score.

Each cyber reasoning system in the competition had obvious flaws, many of which could be addressed easily. We surmise that if the competition were run again, perhaps with a different scoring system, the results might be quite different. In the end, Xandra performed admirably well, finishing second in the overall competition. Xandra was a bit overprotective, hardening services when it was not necessary. It also had a few bugs that caused it to break functionality. Despite these issues, Xandra was best in security, second best in balancing availability and security, and fourth in PoV generation.

The Cyber Grand Challenge provides a tantalizing glimpse into a future where vulnerabilities are discovered and remedied automatically by autonomous systems. Our team continues active research programs in reverse-engineering binaries, fuzzing, concolic execution, binary rewriting, and developing security transformations. Our ultimate goal is to scale CGC-developed techniques to real-world software and secure our cyber infrastructure. ■

Acknowledgments

We would like to thank our families, who provided unrelenting patience and support. Thank you to Kevin Scott and LinkedIn for donating a high-performance cluster. This material is based upon work supported by the Air Force Research Library (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-14-C-0110, FA8750-15-C-0118, and FA8750-15-2-0054. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL or DARPA.

References

1. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," *Proc. Network and Distributed System Security Symposium*, 2016.
2. W. Hawkins et al., "Zipr: Efficient Static Binary Rewriting for Security," *Proc. Dependable Systems and Networks*, 2017.
3. M. Abadi et al., "Control-Flow Integrity Principles, Implementations, and Applications," *ACM Transactions on Information and System Security* (TISSEC 09), vol. 13, no. 1, 2009, p. 4.
4. L. Davi et al., "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," *Proc. 23rd USENIX Conference on Security Symposium* (SEC 14), USENIX Association, 2014, pp. 401–416; http://dl.acm.org/citation.cfm?id=2671225.2671251.
5. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 05), ACM, 2005, pp. 190–200; http://doi.acm.org/10.1145/1065010.1065034.
6. A. Slowinska, T. Stancescu, and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," *Network and Distributed System Security Symposium*, 2011.

Anh Nguyen-Tuong is a principal scientist at the University of Virginia. Contact him at an7s@virginia.edu.

David Melski is a chief technology officer at GrammaTech. Contact him at melski@grammatech.com.

Jack W. Davidson is a professor of computer science at the University of Virginia. Contact him at jwd@virginia.edu.

Michele Co is a research scientist at the University of Virginia. Contact her at mc2zk@virginia.edu.

William Hawkins is a PhD student at the University of Virginia. Contact him at whh8b@virginia.edu.

Jason D. Hiser is a principal scientist at the University of Virginia. Contact him at hiser@virginia.edu.

Derek Morris is a software engineer at Microsoft. Contact him at dmorris321@gmail.com.

Ducson Nguyen is a software engineer at GrammaTech. Contact him at dnguyen@grammatech.com.

Eric Rizzi is a software engineer at GrammaTech. Contact him at erizzi@grammatech.com.

The Mayhem Cyber Reasoning System

Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson | ForAllSecure

Mayhem, one of the first generation of autonomous computer security bots that finds and fixes vulnerabilities without human intervention, won the DARPA Cyber Grand Challenge in August 2016. In this article, we detail Mayhem's creation and look forward to a future where autonomous bots will radically improve computer system security.

We are losing the battle against criminals who break into our computer systems. The reason: we rely only on humans to find new vulnerabilities and fix them. What if, however, we did not need to rely on human effort alone to find vulnerabilities? What if we could build intelligent computer bots that can find and fix vulnerabilities? And what if these intelligent bots could reduce the time to identify and remediate a vulnerability from human time scales—days, months, or years—to computer time scales of milliseconds?

The art of securing computer systems and processes against malicious actors has a broad frontier. DARPA's Cyber Grand Challenge (CGC) explored a new technology on the forefront of cybersecurity: the cyber reasoning system (CRS). A CRS is a fully autonomous system that takes complete responsibility for defending a set of software services. CRSs competing in the Cyber Grand Challenge demonstrated techniques in all core cybersecurity areas, including automatically developing firewall rules to stop attack traffic, analyzing programs to find bugs before an attacker, and patching vulnerabilities in compiled programs without any access to source code. Software defenses that might take human analysts hours, days, or weeks to develop and test were all deployed at machine speeds, on the order of tens of seconds, with no humans in the loop. This first generation of CRSs offers hope for an automatic first line of defense against attacks conducted using novel exploits on large scales at machine speeds, in addition to greatly tightening a defender's response time to other, more typical attacks.

We are the authors, designers, and developers of Mayhem, the winning CRS in the Cyber Grand Challenge, and this article discusses the design and implementation of Mayhem, lessons learned while preparing for the challenge, and our key takeaways from the competition.

DARPA's Cyber Grand Challenge

The CGC was a competitive, symmetric game played by seven fully autonomous CRSs and moderated by a "referee" scoring system. Here, we provide a brief overview of the game mechanics as well as the main ways in which CRS systems could attack and defend. We refer the interested reader to a presentation by the program manager[1] for a more comprehensive presentation of CGC.

Game Structure

The CGC was structured similarly to a networked "capture-the-flag" competition. Players raced to find, exploit, and fix software bugs in their services in an adversarial environment in real time. The competition started

 Copublished by the IEEE Computer and Reliability Societies

when the network was brought up. Each CRS was responsible for a networked server running an identical set of vulnerable software services. The referee served up previously unknown vulnerable software throughout the game.

The referee brokered all communications during the game. Each CRS was on a different and isolated network so that CRSs could not communicate with each other. In addition, a CRS did not have direct control over the server running the vulnerable software services and could only interface with that server through the referee.

To score points (or avoid losing them), players had to perform three main tasks during the competition. First, they had to protect their software from adversaries by finding and patching vulnerabilities. Second, they needed to keep their software available, functional, and efficient. Third, they needed to exploit vulnerabilities in their adversaries' software.

Players scored points by keeping their services running and exploiting the software of other teams, and lost points by damaging their own software or having their systems exploited. All exploit attempts were routed through the referee and mixed in with the referee's tests, which checked that patches to the defended software had not broken its functionality or performance. At the end of each five-minute round, the referee calculated the scores for all the players based on which services they kept running, how performant and functional those services were, and whose services were exploited by whom.

All CGC services were run on the DECREE architecture: a Linux variant with seven system calls and a custom executable format very similar to the binary format commonly used in Linux. This kept the scope of the engineering effort within the reach of small teams, while also keeping the platform close enough to Linux for tools to require straightforward engineering to extend to production systems. It also limited the damage teams could do to each other via exploitation.

Attacking

To attack a service, a CRS had to submit to the referee a program, called an *exploit*, that proved the existence of a security-critical software vulnerability in the target service. The submitted "proof of vulnerability" (PoV) program had to either gain code execution within the target service—which, in the real world, corresponds to taking over a server running that service—or manipulate the target service into leaking privileged data, a less versatile attack than hijacking a server but, as demonstrated by the infamous 2014 Heartbleed bug, a potentially serious vulnerability. Attacks against a particular service could be conducted up to 10 times per five-minute round. This meant that attacks did not need to be perfectly reliable, but an unreliable attack thrown many times increased the chances that the target would notice, patch their service, and reflect your exploit. Consequently, reliable and stealthy exploits were encouraged by the structure of the competition.

Defending

Each CRS had two independent methods to defend each of its services: intrusion detection and binary patching. To use the competition's intrusion detection system, CRSs had to write firewall rules in a language similar to the Snort Intrusion Detection System (IDS) rule language and submit them to the referee, which then deployed them in front of a given service. We did not use the IDS functionality, as its performance impact on the protected services was difficult to predict. Instead our defense relied entirely on binary patching—finding vulnerabilities in the services themselves and fixing them at the machine-code level. Typically, when vendors release a security patch, they make modifications to the program's source and recompile. We did not have the luxury of source, and consequently patching correctly was quite challenging.

A pivotal aspect of the CGC was that deployed firewall rules and patches were harshly penalized for disrupting the operation of services. This could occur either by misidentifying legitimate network traffic as an attack, introducing a bug into the service that caused it to malfunction, or degrading the performance of the service. The correctness and performance of a service were reflected by an "availability score" calculated by the CGC referee. Availability scores began to drop steeply at 5 percent overhead compared to an unpatched service with no firewall rules. Despite this high bar, after optimizing our patches for performance, most of the patches we deployed suffered negligible performance penalty: out of the seven patches that were scored by the referee, our average overhead was 2.64 percent for time and 0.98 percent for memory. All had perfect functionality.

In addition, to represent the cost of distributing a patch across a defended network, whenever a CRS deployed a patch or a network filter for a service, the referee took that CRS's copy of that service down for a round, effectively giving a 0 availability score for one round every time you patched. This made deploying a buggy patch very expensive—you lost a round of points when you deployed it, lost points every round that it was deployed, and then lost another round of points when you deployed a replacement patch.

These tradeoffs are not unique to the Cyber Grand Challenge. Network administrators commonly forgo applying security patches until they know the patches will not disrupt vital services. Vendors are careful to mention in security announcements how severe software vulnerabilities are as well as whether the bugs are known to be exploited in the wild so that their customers can decide how detrimental not patching would be. Today, this data is gathered by humans, but an automated solution able to

optimize service availability without sacrificing security would be immensely valuable.

Scoring

Because the scoring algorithm drove many of our design decisions, we briefly describe it here. For each service, the score was calculated each round for a given CRS as follows:

$$Score = Availability \times Security \times Evaluation$$

Availability is the measure of performance and functionality described above, security is 1 if any adversary proved a vulnerability in this CRS's instance of the service during this round and 2 otherwise, and evaluation is $1+\frac{x}{N-1}$ where x is the number of competitors successfully attacked on this service by the CRS, and N is the number of CRSs participating (which was seven).

The score for a given round is the sum of all the services' scores. Similarly, the total score of a CRS is the sum of its scores over all the rounds.

Mayhem Defense

Mayhem confirms and patches software flaws differently from a human developer or security analyst. When analyzing a service, Mayhem confirms a software flaw only when a test case that causes the service to crash or otherwise exhibit potentially exploitable behavior is available. Mayhem has no access to the original source code or the specification of the application (while both are typically available to the developer). Mayhem only knows how a service actually behaves and that services should not crash or be exploitable. Thus, patches deployed by Mayhem aim to render identified software flaws unexploitable.

Mayhem patches are based on runtime property checking. For every identified flaw, Mayhem inserts introspective checks into programs, similar to assertions that a developer might add. These checks verify runtime properties that are exceedingly likely to be true for a correctly operating C or C++ program, and not likely to be true for a CGC service that is being exploited. Specifically, a Mayhem patched service verifies at various points that it would not access memory at, or branch to, invalid or unusual addresses. Failing a check leads to safe termination of the service, preventing exploits from successfully completing.

Although these checks could be placed in thousands of locations throughout a service's code, too many such assertions reduces performance. Mayhem places checks along any code paths that it found crashing test cases for, and also at a few heuristically identified points. Furthermore, Mayhem uses formal methods to elide checks that are proven unnecessary.

Mayhem is also capable of defending its services using several established exploit mitigation techniques: stack canaries, address randomization, data execution prevention, and a simple tag-based control flow integrity (CFI) scheme. When utilized by a compiler, these mitigations can obviate entire classes of exploits, while rarely interfering with correct program behavior. Unfortunately, although introducing these mitigations into compiled services after the fact is possible, the added code may affect performance and/or functionality. Mayhem relies on its test case generation capabilities to verify that the inserted mitigations do not break the performance or functionality of services and to fall back to less comprehensive mitigation techniques when the performance overhead is unacceptable.

Binary patching is difficult. Even after deciding which checks and mitigations to insert where, Mayhem has to insert the patches and generate a functional, performant binary. Lacking source code, Mayhem patches services by directly editing the assembled processor instructions within the program. This is a nontrivial task because simply inserting instructions into the middle of a program would break the various fixed addresses and relative offsets with which a program refers to itself. This is similar to how inserting pages into a printed book would render the page numbers in the index and table of contents inaccurate, but much more critical and much harder to fix. In the next section, we discuss the core approaches Mayhem follows for tackling the practical issues of binary patching.

Binary Patching Techniques

Mayhem has two techniques for inserting patches (see Figure 1). Its preferred method, full-function rewriting (FFR), approaches the above challenge directly. FFR inserts patches into the middle of the program, and then attempts to repair the issues this causes by adjusting all addresses and offsets within the program accordingly. This is a tricky process, and our methods were not perfect: we relied fully on static analysis, which resulted in malfunctioning patched services on 0.57 percent of our test binaries. To ensure maximum reliability, we tested our methods daily on binaries released by the organizers and compiled across a variety of optimization levels during development, for a total of 293 test binaries. In addition, during the competition Mayhem was careful to not deploy FFR-patched binaries until it had verified that they functioned as intended on the referee's network traffic and generated test cases.

The other patching technique Mayhem employs is more conservative. Injection multipatching (IMP) avoids comprehensive modifications to programs by not inserting patches into the middle of a program. Instead, IMP substitutes a branch instruction into the program at each location to be patched. These branch instructions jump into a custom section appended to the end of the

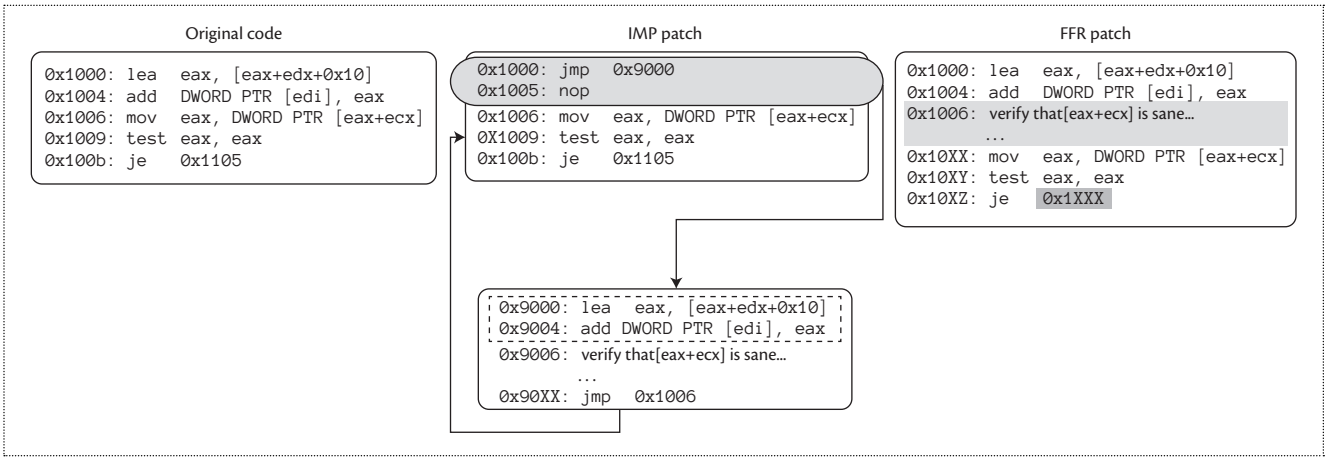


Figure 1. Injection multipatching (IMP) versus full-function rewriting (FFR) techniques. IMP clobbers some instructions to jump out to the patch body, located in a section at the end of the program. FFR inserts patches directly inline, but all program addresses are changed; many control flow instructions and code pointers must be updated accordingly.

program, which contains the instructions implementing the patch—along with any instructions displaced by the branch. To continue the book metaphor, this is like overwriting a sentence with "See Appendix A." IMP is more reliable than FFR because it leaves the majority of the program untouched, but the extra jumps to the added section increase the performance overhead of patched services.

Network Intrusion Detection

Mayhem did not use the network intrusion prevention system available to CRSs. Signature-based defenses like network filtering have trouble generalizing to defend against polymorphic exploit variants, and they might spuriously trigger on inputs that do not pose a threat to the service. Furthermore, they can inform other teams about the nature of the identified flaw. Mayhem's patches are much less susceptible to these failure modes, because they catch attacks as they exploit critical software flaws, do not interfere with normal service operation, and include obfuscation.

Mayhem Offense

Mayhem's offensive capabilities are based on generating test cases that demonstrate flaws in a target service. While humans tend to discover exploitable vulnerabilities by first identifying potentially flawed sections of code and then figuring out how to trigger that flaw, Mayhem needs to "see" a flaw occur before it can identify or attempt to exploit it. Mayhem makes up for this deficiency by generating and analyzing thousands of inputs per second, each of which could trigger a vulnerability.

Mayhem generates these test cases using a combination of gray box fuzzing and white box symbolic execution techniques. Other competing teams also combined the two.[2]

Symbolic Execution

Symbolic execution[3] is a method for reasoning about the behavior of a program by building logical formulae that represent the program. The inputs to the program are treated as variables, and any value computed by the program—such as return values, or branch conditionals that affect choices the program makes—is represented as a function on these input variables. Using an SMT solver, these functions may be solved for a desired output value to calculate an input test case that ensures the program will reach a desired state.

Mayhem's symbolic execution engine[4] manipulates the branch conditionals within a program. By generating inputs that will result in the program having different values for the conditionals it computes, Mayhem forces the program to execute in a variety of different ways. This exercises different areas of code within a program under a variety of circumstances, exposing software flaws.

The specific strategy Mayhem uses to generate test cases is known as *concolic execution*. Starting from an arbitrary initial seed test case, Mayhem traces the concrete (nonsymbolic) execution of the target program on that test case, while simultaneously building symbolic formulae for values derived from the input. Every time Mayhem encounters a conditional branch, it uses the formula for the condition to produce a modified input test case that takes the other direction on the branch. We call this production of modified test cases *forking*. Mayhem also forks for values other than branch conditionals, such as pointer values—this generates new test cases for which the program accesses memory differently. As Mayhem traces the seed test case, it forks many times. After the program completes execution of the seed test case, Mayhem repeats the tracing process using the forked test cases as new seeds.

Repeatedly tracing all forked test cases as new seeds leads to exponential growth of the number of seeds, rapidly overwhelming a system's capability to process them all. This "path explosion problem" is not specific to concolic execution; it arises from the fact that even simple programs tend to have an exponentially large number of possible states to explore. Mayhem employed a novel technique called Veritesting,[5] which mitigates path explosion by merging similar seeds together; processing a single merged seed achieves exploration equivalent to processing each of its constituent seeds.

During the CGC competition, our symbolic execution component found crashes for 62 of the 82 services, including five services that fuzzing did not crash.

Fuzzing

Our fuzzing was largely guided by the open source project American Fuzzy Lop (AFL; http://lcamtuf.coredump.cx/afl). This tool was designed to do intelligent, fast fuzzing using compile-time instrumentation to measure edge coverage in programs. Here, we describe some differences between AFL and our fuzzing.

As we did not have the source code for our target services, Mayhem uses alternative techniques for gathering coverage information. The first technique uses our FFR patching infrastructure: we inserted a series of patches that implemented the necessary code coverage tracking and reporting mechanisms. Although this technique was complicated and fragile, it allowed us to add many other useful features. For example, 32-bit comparisons were rewritten as four consecutive 8-bit comparisons to allow the fuzzer to make incremental progress with its random exploration and mutation, without needing to randomly guess the correct 32-bit value. In addition, initialization done by the binary could be skipped and analysis could be deferred until the program began reading user input, greatly accelerating fuzzing. In the end, FFR-assisted fuzzing allowed us to run the binary with negligible performance impact. In the rare event that FFR was not able to successfully patch the program, we fell back to a modified version of QEMU, as used by the AFL fuzzer. Although this was a safe option, it suffered a 100 to 900 percent performance hit.

During the qualifying event for the CGC, fuzzing helped us find bugs in 65 of the 131 binaries in 24 hours. Based on write-ups from other teams, this suggests that our fuzzer was far more effective than any fielded by any other team. Furthermore, continued improvements to our fuzzer between the qualifying and final events roughly doubled our bugs per hour; it found crashes in 92 of those 131 binaries from the qualifying event in a 16-hour period just before the final event. During the final CGC event, fuzzing found crashes in 62 of the 82 services (five of which were not found by symbolic execution).

Both of these techniques—symbolic execution and fuzzing—make use of code coverage metrics, particularly edge coverage. Coverage-guided program exploration searches for test cases that reach parts of the target program that no other test case has reached before. By analyzing the frontier of test cases that achieved new coverage, more and more parts of the code will be explored over time. By using the same coverage metric for test case prioritization in our fuzzer and symbolic executor, we minimized the amount of work duplicated between the two components. Working together, the two systems found crashes for 67 of the 82 services, just over 80 percent. Of the 57 crashes found by both components, symbolic execution found 27 before fuzzing, while fuzzing found 30 first.

Automatic Exploit Generation

After finding bugs, Mayhem had to exploit them. The Automatic Exploit Generation (AEG) component took in a CGC program (either an unpatched challenge binary or an opponent's patched binary), an opponent's network filter, and a crashing input and attempted to generate a PoV. To generate PoVs, we used a combination of black box and white box techniques. The black box AEG component inferred the relation between the input string and the crashing state through input mutations, and then generated an input that demonstrated the desired condition for a PoV (EIP control or a leak of sensitive data). The white box AEG component symbolically executed the crashing path and attempted to satisfy the constraints necessary for exploitation. If the constraints were satisfiable, Mayhem used the solution to emit a PoV. In addition, those constraints might depend on the data sent by the service. For instance, a service might expect a client to replay a nonce it sent or to solve a simple mathematical CAPTCHA. In those scenarios, the PoV generated the input on the fly by querying an embedded SMT solver.

All generated PoVs were tested in the DECREE environment to measure their reliability before deployment. At the end of the competition, Mayhem's database contained working PoVs for 18 services. This represents a lower bound on what we actually found, as our system deleted most (and sometimes all) PoVs for a challenge after that challenge was removed from the competition by the referee in order to free up disk space on the database node.

Mayhem Strategy

Strategy was an essential component. CRSs had to make strategic decisions throughout the game: Which binaries to patch? Which patches to deploy? Which teams to attack, and with which exploits? Where should limited resources be spent?

A bad strategy could ruin the winning chances of an otherwise good CRS. A single bad decision could have disastrous consequences. Indeed, as all the services were running on a single host, deploying one patched binary (or network filter) consuming too many resources could

affect all the services in play. In fact, this scenario happened to at least two CRSs, and might have cost one team third place and $750,000.

Patch Scoring and Selection

To avoid deploying bad patches, we took maximum advantage of the opportunities presented by the organizers to observe their scoring system and deduced which performance metrics were being used where the documentation was ambiguous. We also developed a distributed testing platform, which replayed seeds from the network against both our patched binaries and the unpatched versions to calculate our overheads and verify that we retained functionality. This allowed us to accurately predict how our patches would score before we deployed them, and we never fielded a functionality-breaking patch.

In addition to guaranteeing performance and functionality, we also had to balance uncertain security against known performance. Whereas our crash-specific patches were performant, they did not protect against vulnerabilities we had not yet found. Our binary hardening techniques could protect against unknown exploits, but they carried performance costs, and we often generated several variants with decreasing security before finding one with acceptable performance. In the end, we generated a priori estimates for the security scores of each type of patching and used these and our observed performance scores from testing to generate estimated scores for our patched binaries. When Mayhem decided to deploy a patch, it picked the one that it projected would score highest—a simple patch selection strategy, ignoring adversary behavior, but effective nonetheless. Given reasonably effective patches, choosing when to deploy was the critical part of the patching strategy.

Because the referee made deployed patches available to other CRSs during the competition, another option for patch selection, which we rejected, was "stealing" and redeploying other teams' deployed patches. Although this might be tempting if another CRS clearly had stronger patching capabilities, this exposes you to back doors, where an opponent could send your copy of his deployed patch a particular input and it could give him code execution. Because we added back doors to our patched binaries, we believed any adversary capable of patching better than us would probably do the same. Automatic back door removal would have required deep analysis of an adversary's patches, which might have exposed us to exploitation (as we discuss later), and so we did not consider it worth the risk. Because we never successfully exploited a back door against another team's service, we believe other teams reached the same conclusions about patch stealing.

Patching Strategies

Deploying patched binaries in the CGC was risky. Aside from the possibility of degraded availability, the patching process incurred one full round of downtime. Leaving binaries unpatched was dangerous, but patching a binary that was never going to be exploited was a straight loss of points. Deploying a poorly performing patch might also have been better in the long run than being constantly exploited. To address these nuances, we examined many possible patching strategies before finalizing on one.

Several naive patching strategies presented themselves, the most obvious of which was "always deploy as soon as you've generated a crash-agnostic patch you think has reasonable performance." Shellphish chose this strategy. Unfortunately, this strategy has some weaknesses. Patching everything as soon as possible led to large patching downtime penalties, and in several cases, Shellphish's patches performed worse than expected, leading to further penalties. In addition, some binaries were never exploited, so leaving the unpatched service up was optimal—as this avoided any downtime penalty and any risk of breaking functionality or performance. Even if a service was eventually exploited, if you generated and selected a good patch beforehand and deployed it promptly when exploitation began, you were only down one turn of exploitation points compared to the naive strategy. Detecting exploitation, however, was nontrivial due to the limited information available from the referee.

Although simple strategies are robust, we chose a complex strategy based on a Bayesian classifier. This allowed us to make informed inferences about which services were being exploited by other teams using all the data available to us. We used information including our points per round, what sort of test cases we had uncovered from our own bug hunting, and whether we had seen traffic crashing each service as observable values that were fed into a classification system with hard-coded initial Bayesian priors.

This system worked well for several reasons. Many of the relevant quantities to calculate the conditional probabilities were directly observable from the game state, giving us good sensors on which to base our measurements. We could also calculate many probability estimates in multiple independent ways and take the geometric average. This allowed us to turn several noisy probability estimates into a higher-accuracy estimate. Overall, this resulted in Mayhem being able to make many intelligent choices. For example, if teams attempted to trick our system into believing that it was under attack by sending "fake" exploits that crashed our service but did not actually exploit services, Mayhem would start treating crashing test cases as an unreliable indicator of being exploited (because our other sensors would not corroborate us getting exploited). This meant Mayhem could adapt to changing and unknown environments, which is a necessity in adversarial settings. The downside to this was that Mayhem's patching strategy was fundamentally reactive. Although it might detect quickly that it was being

exploited, it would never patch a service before it believed that there was an exploit in the wild.

Once we had an estimate for the probability that a service was being exploited, we compared it to a threshold to decide if we should deploy a patch. Rather than setting a fixed probability cutoff for when we should patch a service, Mayhem instead dynamically adjusted this threshold. This allowed us to adapt across a variety of possible strategic situations—if we observed that teams patching many or all services were doing well, we could become more aggressive in our patching to match them, but if we observed that high-scoring teams were conservative with their patching or that we were losing points for patching too aggressively, we could adjust this threshold.

Finally, once we determined that a service was probably under attack and we made the decision to patch, we evaluated the scores of our possible replacement binaries. In addition to our continuously updated performance estimates for our patches, we factored in estimates of how much longer the service would stay in the game and how badly a round of downtime would impact us (to avoid, for example, taking a round of downtime if the service was only expected to remain in play for one more round). After these calculations, if we decided that it would benefit our score to patch a service, we did so.

Anecdotally, we observed many cases in which Mayhem made "good decisions." We tested Mayhem's decision-making process through a number of practice CGC games and were often surprised by when it chose to patch or leave services unpatched. As humans we frequently make decisions irrationally, treating "worrying" indicators such as a crash in one of our services as strong indications of exploitation, even when statistical evidence in the game suggests these indicators are untrustworthy.

Offensive Strategy

CRSs had to make many offensive decisions throughout the competition. At each round and for each service in play, systems had to decide which teams to attack and which exploits to use. For instance, a CRS might not want to send an exploit to another system that is good at reflecting exploits by extracting them from the network tap.

Strategy is a function of the objective function. The CGC was no exception, and our strategy was specifically tailored to the CGC scoring. In particular, the scoring function penalized insecure binaries more than it rewarded successful attacks. Therefore, we concluded that the optimal offensive strategy was to send exploits, when available, to all teams at all times.

However, Mayhem still attempted to minimize reflection attacks by generating stealthy exploits. For instance, information disclosure attacks are harder to detect than a crashing input. Mayhem ranked exploits by stealthiness and submitted the stealthiest one.

Mayhem Autonomy and Counter-Autonomy

The CGC was a fully autonomous competition. No human intervention was allowed once the game started. The machines were entirely on their own, and the CRSs needed to maintain themselves for the duration of the competition in an adversarial environment.

One of our largest system resilience concerns was *counter-autonomy*, that is, adversaries sending us malicious input to disrupt the different components of our CRS (as opposed to exploiting the services provided by the referee). Here, we describe how Mayhem monitored itself to maintain uptime. We then discuss the different techniques that could be used to disrupt the correct operation of an autonomous CRS. Finally, we go over countermeasures to adversaries with counter-autonomy capabilities.

Autonomy

During development, we quickly realized that Mayhem would encounter scenarios that could not be predicted, because the CGC would be our first time handling inputs from real and motivated adversaries. Therefore, we worked under the assumption that components would go down. To simulate unexpected failures and challenge Mayhem's resiliency, we developed Chaos Monkey (https://github.com/Netflix/chaosmonkey).

Due to the full autonomy requirement, significant resources were spent on ensuring reliability and automatic recovery from failures. Our autonomy strategy relied on three main techniques: defensive programming, liveness monitoring, and redundancy.

We designed our components defensively. A significant portion of Mayhem's code consists of error handling, with a heavy use of timeouts to ensure progress. All components run in infinite loops with cron jobs to bring them back up if they die. We also have cron jobs to kill leaked processes and delete leaked temporary files.

To improve autonomy, we designed the Medic, a component for monitoring liveness and health of processes, VMs, and physical machines and restarting them as necessary. Liveness and health had different meanings for different components. For processes, that meant being alive and making meaningful progress, like testing new network inputs rather than wedging in a tight loop testing one pernicious input repeatedly. For VMs and physical machines, that meant that processes running on them were healthy and that CPU, memory, and disk usage were within expected bounds.

We added redundancy to many of our CRS components and ran them on multiple hosts concurrently. We ran multiple copies of the Medic monitoring each other and widely distributed our network seed testing, fuzzing, and exploit generation subsystems. For components that could not be duplicated easily, we designed failovers.

If one failed, the others would activate seamlessly. For instance, we put a lot of effort into securing our expected single point of failure: our database. We set up a hot/actively-synced replica of our database on another host, wrote automated failover routines, and tested this functionality extensively. We also had a failover for our patching system and our communications with the referee.

The only one of these reliability measures that we know fired during competition was the Medic, which killed the component downloading data from the referee to our database because it was running abnormally slowly. Unfortunately, killing that component did not solve the problem but made it worse, by increasing the backlog we had to pull down from the referee.

Counter-Autonomy

Counter-autonomy aims to hinder, deceive, or shut down an adversarial autonomous system's operations. In the CGC, CRSs could see traffic sent to their networked software as well as the patched binaries and network filters deployed by their adversaries. Those inputs should be considered malicious. Carefully crafted binaries or network packets could potentially slow down or take out an autonomous system for the rest of the competition. Counter-autonomy techniques were used by several teams during the CGC, and we discuss some of them here.

One of the least damaging counter-autonomy attacks simply slows down the target CRS. For instance, in the CGC, patched binaries from several teams[6] could detect when they were run under popular dynamic analysis frameworks like Intel PIN[7] or QEMU.[8] Under analysis, these binaries behaved differently, hindering the autonomous system by wedging the analysis system, wasting CPU cycles in tight loops, or allocating gigabytes of memory. Deploying patched binaries that varied their behavior when run in different environments was somewhat risky, as we had limited guarantees about the environment in which the binaries' performance would be measured for scoring purposes. Nevertheless, some teams considered the risks worth the potential benefits.

More damaging counter-autonomy techniques attempt to destroy rather than slow down the adversary's system. The goal of this sort of attack is to directly exploit components of the autonomous system itself. For instance, during the CGC, we assumed that most CRSs would replay packets received from the network tap against the unpatched version of the binary to detect crashes and exploits. On a crash, CRSs would likely perform additional analysis under a dynamic analysis framework. A malicious test case could exploit the unpatched binary to gain code execution, and then exploit the dynamic analysis framework to run code directly on the machine doing analysis. If no countermeasures were in place, these attacks were potentially game-ending. For example, a particularly nefarious payload could connect to the unauthenticated power API to turn off the adversary's machines for the rest of the game. Another option was to scrape the disk for credentials to the referee API and submit broken patched binaries for all the services in play, preventing the target from scoring any points. Although the CGC rules prohibited attacks of this kind, as security researchers, we planned for the worst. We took precautions to prevent these attacks on our system and to minimize our risk.

Finally, more elaborate and subtle counter-autonomy techniques involve deception in order to trick adversaries into making costly strategic mistakes. These are perhaps the most difficult to defend against. Attacks of this kind require predicting how adversaries will react to certain behaviors and using those reactions to your benefit. For instance, we predicted that once a service started crashing, a CRS might infer, perhaps incorrectly, that the service was being exploited, and consequently patch it. Due to the scoring algorithm, patching too early was a net loss; if no team was exploiting you, deploying a patch meant taking a performance and downtime penalty for no gain. Therefore, Mayhem sent harmless crashes like null-pointer dereferences to services we could not exploit, and it led some CRSs to make strategic mistakes in their patching decisions during the competition. In general, however, the structure of the competition made deploying deception difficult, as we had no data on the strategy and behaviors of our opponents. In a multicompetition series, however, deception would become important.

Counter-Counter-Autonomy

An autonomous CRS requires counter-autonomy countermeasures to be resilient in an adversarial environment. Consequently, we designed several protections in Mayhem. At a high level, our countermeasures are split into three categories: minimizing the attack vectors, fault isolation, and sandboxing.

Mayhem minimizes analyses conducted on attacker-controlled inputs. In particular, Mayhem never leverages static analysis and limits use of dynamic instrumentation frameworks on an adversary's patched binary. This was a conscious design decision: we did not think the benefits would outweigh the risks. On the one hand, this handicapped our CRS, as Mayhem could potentially have found more exploits by analyzing adversaries' patches. On the other hand, doing so would have put us at a greater risk of being exploited and taken down.

Mayhem did, however, run adversaries' patched binaries (albeit without analysis) to check that our exploits worked locally before deploying them to the network. This was a calculated risk, but one that we minimized with isolation. To limit the damage from a potential attack, Mayhem had one dedicated virtual machine per adversary, and it was the only machine on which we ever ran

their binaries. The effects of an attack would be limited to that one VM (assuming no ability to escape from the VM) and would not affect our analysis of other teams or the unpatched services.

Finally, we needed to consider malicious test cases from the network. Network traffic contained test cases sent by the referee to check availability, and those test cases were critical for both our offense and our defenses—they provided initial seeds for our fuzzing and symbolic execution, and they were also ground-truth for the intended functionality of each service. However, this network traffic potentially included malicious traffic from adversaries. Therefore, we handled data from the network (as well as data derived from it) with care. We hardened our dynamic analysis frameworks against public and homemade exploits, and we sandboxed our analysis processes with a combination of system call whitelists and limits on CPU and memory.

Mayhem's Autonomy in the CGC

Despite all the techniques mentioned above, we are sad to say that Mayhem still ran into issues during the competition. We failed to account for a failure mode in the component downloading round data from the referee. Although downloading a single round usually took less than 30 seconds, it got much slower starting at round 27 (out of 96). Some rounds took more than 18 minutes to download fully. As rounds were only five minutes long, this component therefore fell behind, and Mayhem did not have access to the current state of the game. Our CRS was essentially playing the game in the past: it was analyzing binaries after the referee had removed those services from the game, unaware that new services had replaced them.

Luckily for us, the number of exploits thrown by other teams decreased significantly starting around the time this issue occurred, and therefore Mayhem's score was not as negatively affected as expected. This lack of exploits made leaving the unpatched binaries up a surprisingly good strategy; it avoided downtime and performance losses. As a result, Mayhem held onto its lead from the first 27 rounds and still won first place. Interestingly, Mayhem recovered near the end the game and landed an exploit in the last round.

The Cyber Grand Challenge shows hope for technologies that were recently considered pure science fiction. With some of the top researchers in academia and industry working for two years toward a goal of automating large parts of software and network security, many hurdles have been overcome. ForAllSecure has always believed that these technologies would transfer to real-world applications, and we are excited to continue development to make automated security practical for real networks and software applications. Automated bug finding, patching, and software testing will help scale up security, hopefully allowing us to catch up to the blistering pace of modern technological development and ensure a safe computing landscape for everyone. ■

References

1. M. Walker, "Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge," *Proceedings of the 24th USENIX Security Symposium*, 2015.
2. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," *23nd Annual Network and Distributed System Security Symposium* (NDSS 16), 2016.
3. E.J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.
4. S.K. Cha et al., "Unleashing Mayhem on Binary Code," *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.
5. T. Avgerinos et al., "Enhancing Symbolic Execution with Veritesting," *ICSE*, 2014.
6. "Cyber Grand Shellphish," Shellphish, phrack.org, 2017; www.phrack.org/papers/cyber_grand_shellphish.html.
7. C.-K. Luk et al., "PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
8. F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," *Proceedings of USENIX Annual Technical Conference*, 2005.

Thanassis Avgerinos is a cofounder of ForAllSecure. Contact at thanassis@forallsecure.com.

David Brumley is the CEO and cofounder of ForAllSecure and a professor at Carnegie Mellon University in ECE and CS. Contact at dbrumley@forallsecure.com.

John Davis is a software engineer at ForAllSecure. Contact at jedavis@forallsecure.com.

Ryan Goulden is an engineer at ForAllSecure. Contact at ryan@forallsecure.com.

Tyler Nighswander is a bidirectional engineer at ForAllSecure. Contact at tylerni7@forallsecure.com.

Alex Rebert is a computer security researcher and cofounder of ForAllSecure. Contact at alex@forallsecure.com.

Ned Williamson is a software engineer at ForAllSecure. Contact at ned@forallsecure.com.

The Past, Present, and Future of Cyberdyne

Peter Goodman and Artem Dinaburg | Trail of Bits

Cyberdyne—a distributed system that discovers vulnerabilities in third-party, off-the-shelf binary programs—competed in all rounds of DARPA's Cyber Grand Challenge. Since then, Cyberdyne has been successfully applied during commercial code audits. We describe its evolution and implementation as well as what it took to have it audit real applications.

The Trail of Bits cyber reasoning system, Cyberdyne, is an automated and distributed bug-finding system. It competed in all rounds of DARPA's Cyber Grand Challenge (CGC), first with the Trail of Bits team, and then as the bug-finding arm of one of the finalists. Since then, it has been successfully used to audit shipping software libraries and for commercial code audits.

Cyberdyne discovers and exploits memory access violations and information disclosure bugs. Like most CGC competitors, Cyberdyne applied two complementary techniques for finding these kinds of bugs. The first technique, fuzzing, repeatedly executes a program on inputs generated by mutating a common seed input. The second and more complex technique, symbolic execution, produces inputs that exercise all feasible program paths.

This article is divided into two separate but equally important halves. The first half of this article will describe Cyberdyne's unique features and approaches. Cyberdyne evolved over the course of the competition into a production quality bug-finding engine. Each component of Cyberdyne is a unique artifact, whose designs were motivated by observations and experimentation.

The second half of this article answers the big question that everyone had after the CGC: What's next? For Cyberdyne, the next step was auditing Linux programs for security vulnerabilities. Despite the deep differences between the OS used for the CGC (DECREE) and Linux (like the lack of files or threads), the vast majority of Cyberdyne could be reused to automatically identify bugs in real Linux applications. We will discuss how Cyberdyne performed the first paid automated security audit, and the challenges of automated security audits. We conclude this article with a discussion about the future of automated bug-finding systems and how automated security assessments will improve software quality and security.

Building Cyberdyne

Architecture

Cyberdyne is a distributed system that tries to prove that a target program has memory access violation bugs or information disclosure bugs. Instead of attacking this problem head-on, Cyberdyne implements a genetic algorithm to produce inputs (genes) that maximize the amount of code executed (fitness function) by the target program. If these inputs crash the target program, Cyberdyne can show the program has memory safety bugs.

1540-7993/18/$33.00 © 2018 IEEE Copublished by the IEEE Computer and Reliability Societies March/April 2018

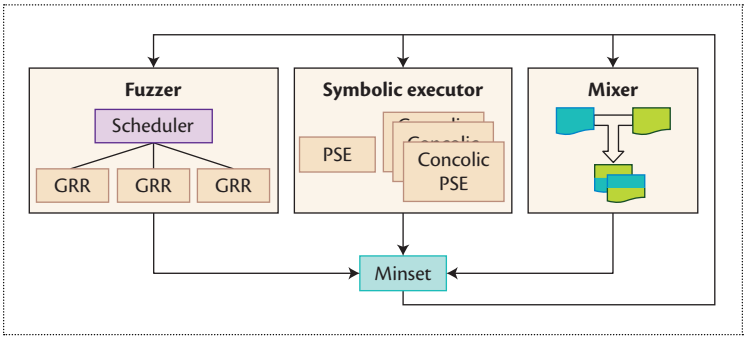


Figure 1. The architectural diagram of a Cyberdyne node.

Nodes in a Cyberdyne cluster mostly operate in isolation, with each node focusing on finding vulnerabilities in one or more target programs. When there are more nodes than target programs, nodes cooperate to share the workload. Cooperating nodes self-configure to use different metrics for bug-finding progress in an attempt to inspect the most target program states.

As shown in Figure 1, each node in a Cyberdyne cluster is composed of four main services:

- the fuzzer,
- the symbolic executor,
- the mixer, and
- the minset.

These services share data and communicate with one another over a shared Redis server. The first two services, the fuzzer (GRR; https://github.com/trailofbits/grr) and symbolic executor (PySymEmu running in symbolic and concolic modes; https://github.com/trailofbits/manticore) are bug-discovery tools. The fuzzer performs input mutation; the symbolic executor synthesizes new inputs. The mixer service performs input crossover, and the minset service implements the fitness function. Fitness is evaluated by how much unique code coverage an input contributes to the set of coverage induced by all generated inputs.

The following sections describe each Cyberdyne service in turn.

Fuzzer

Fuzzing is a software assurance methodology that uses input generation and concrete execution to discover security faults. Fuzzing is widely accepted in the software security industry and is used by large companies such as Microsoft, Google, and Adobe. Any fault produced by a fuzzer represents a real fault in the program. However, the absence of identified faults does not imply the program is bug-free.

Fuzzers work by feeding randomized inputs into a program and determining whether these inputs cause the program to crash or to execute new code. Modern fuzzers instrument the target program and modify inputs based on feedback from prior executions.

Cyberdyne's fuzzer service has two components: the scheduler and GRR. The scheduler orchestrates GRR and decides what, when, and how to fuzz. GRR feeds and mutates inputs to the program, and instruments the program to determine when an input causes a crash.

Scheduler. By default, the scheduler devotes an equal share of CPU resources to fuzzing each program. When a vulnerability in a program is discovered, the node responsible broadcasts this fact to others, thereby reducing CPU resources dedicated to fuzzing that program. This was a prudent choice for the competition, where the goal was to exploit as many programs as possible.

The scheduler operates on the list of inputs supplied by the minset service, which is described later. By focusing on a limited set of inputs, the fuzzer makes efficient use of limited CPU resources. A positive side-effect of using the minset to select fuzzing inputs is that the fuzzer cooperates with other input sources (for instance, the symbolic executor). Inputs in the minset could, and did, have a mixed ancestry: an input may first be created via symbolic execution, then modified several times via fuzzing before crashing the target. One way to visualize cooperation between tools is as a kind of hill-climbing. The fuzzer mutates inputs that are highest on the hill, hoping that the mutation will yield new code coverage even higher up.

The inputs themselves are prioritized for fuzzing by recency. Inputs most recently added to the minset are allotted more CPU resources than those added long ago. This fits with the hill-climbing theme; recent inputs more likely exercise deeper program states (higher on the hill), by virtue of being derived from inputs produced earlier (lower on the hill).

Separating scheduling from mutation allows us to make the scheduler smart and the mutator extremely fast. While the scheduler selects what to mutate, GRR performs the mutation.

GRR. GRR is an emulator with built-in support for mutating inputs and instrumenting program execution. GRR was designed to maximize throughput: a single GRR process can cycle between mutating an input and performing an instrumented execution of that mutated input. In the CGC, a single GRR process executing for one hour would typically perform one million input–execute cycles.

Dynamic binary translation. GRR is a 64-bit x86 program that emulates 32-bit x86 instructions using

dynamic binary translation (DBT). It dynamically decodes a sequence of 32-bit x86 instructions (called a basic block), modifies the decoded instructions (perhaps adding in new ones), and encodes and later executes the new instruction sequence. When executed, the new instruction sequence performs the same operations as the original sequence, but with additional instrumentation (for example, recording what code is executed, intercepting memory accesses).

As a 64-bit program, GRR can use more hardware registers and memory than the original program. This setup simplified translation from 32-bit to 64-bit machine code. The translated machine code could use an additional eight registers guaranteed not to interfere with any registers used by the original machine code. Each of these eight "newly available" registers were given specific meanings in the translated code. For example, the translator rewrote all memory-accessing instructions to use a special "MEMORY" base register that pointed at the base of the emulated 32-bit address space.

Code caching. Existing DBTs (for instance, Intel PIN, DynamoRIO) retranslate the same program every execution. They specialize in translating long-running programs, where the translation cost is amortized over time, and "hot" code is reorganized to improve performance. DBTs avoid retranslating the same code by storing translations in an in-memory cache, and indexing that cache with a lookup table. Fuzzing campaigns like the CGC and code audits require billions of program executions. This means that all code in a target program, even in short-running programs, is hot. This realization motivated GRR's code cache and index persistence feature. Independent executions of GRR need not retranslate the same code.

A unique feature of GRR's persisted cache index is that it permits caching of self-modified or just-in-time compiled code.

GRR's cached translations can also be reused for different analysis purposes. For example, the same cached translations can be used for recording code coverage and memory access interception. This flexibility is achieved using a combination of specific meanings for each additional 64-bit register and a "stub function" mechanism for instrumentation callbacks.

A unique feature of GRR's persisted cache index is that it permits caching of self-modified or just-in-time compiled code. Typical DBT cache indices map the virtual addresses of original instructions to those of translated instruction. GRR's index maps original, 32-bit instruction addresses and a code "version number" to the offset of the translated code within the persisted cache file. The version number is a Merkle hash of the contents of executable memory at the time that the instruction was translated. Modifying the contents of an executable page in memory invalidates its hash, thereby triggering retranslation of any code on that page when it's next executed.

Snapshotting. GRR is actually two programs: the first program takes "snapshots," and the second program emulates executions, using the snapshots as a template for the program's initial state. Snapshotting was motivated by the observation that programs typically execute deterministic setup code prior to receiving input. Snapshotting execution prior to reading external input allows GRR to skip setup code and to ensure such code does not contribute to code coverage.

OS and I/O. GRR emulates programs that interact with DECREE, a custom, Linux-like operating system developed by DARPA for the CGC. GRR is a single-threaded program; however, it can emulate concurrent target programs communicating via sockets. For example, GRR can emulate a server, a client program, or both at the same time. GRR implements a simple round-robin process scheduler, swapping process contexts between system calls.

GRR emulates all I/O in memory and can perform millions of independent input mutations and execution emulations during a single run of the GRR process. This setup makes GRR entirely CPU-bound.

How GRR fuzzes. A typical GRR fuzzing run begins by executing a target program on an initial input file and recording how the target program consumes the input. The recording of all the input entering the program, and how the input arrives, serves as the basis for GRR's input mutation.

GRR mutates input recordings via three different granularities. The smallest granularity, system call, applies a mutation operator to the input bytes of one or more consecutive read system calls. For interactive programs, this often represents byte-granularity mutation. For structured or layered input formats, this can enable mutation of the embedded data. The second granularity, line granularity, compresses the system call recording, combining uninterrupted sequences of read system calls into logical "lines" of input. Other I/O system calls represent interruption points. For some interactive programs, this enables mutation of actual user-supplied "answers" to input prompts. The

last granularity, file granularity, behaves just like other fuzzers: a mutation operator is applied to the initial input as a whole.

The fundamental mutation operators (for example, flipping a bit in each input byte) are implemented as transformations within GRR. For more complex mutations, GRR can embed external mutators. By default, GRR uses Radamsa (an open source input mutation engine; https://github.com/aoh/radamsa) when mutating longer inputs.

GRR's recording and replay can also be used to explore program inputs generated by other sources. One of the key input sources, described in the next section, is the symbolic executor, which is a crucial source of the raw input data used for mutation.

Symbolic Executor

PySymEmu (PSE) is a custom symbolic execution engine written in Python and was originally developed prior to Cyberdyne. We chose PSE because it was easy to understand, extend, and integrate into Cyberdyne. PSE is composed of three main parts: a symbolic CPU, a memory model, and an operating system model.

PSE operates directly on x86 machine code and runs by iteratively decoding and emulating x86 machine instructions. Because PSE reads each instruction from emulated memory, it can handle edge cases like self-modifying code that can stymie other symbolic executors. The PSE memory model is specifically suited for analyzing binaries: PSE divides memory into pages, which can contain a mix of concrete and symbolic bytes, and can be addressed via both concrete and symbolic memory addresses.

PSE's operating system model supports symbolic data as arguments to system calls. In such cases, PSE will fork the analysis to explore all possible concrete states. PSE also implements basic support for multiprocessing.

As with any symbolic execution tool, PSE has to choose from many possible program states to analyze. PSE totally orders all states according to a metric based on the number of input bytes read, output bytes written, and total and unique instructions executed. The same metric is applied to all programs. As a hedge against this metric being a poor choice for a particular program, PSE will randomly choose between the top and bottom five ranked states. Fundamentally, this state-ordering metric provides no deep insights: it was experimentally chosen by trial and error.

What made PSE effective in Cyberdyne was its ability to use GRR program snapshots to simulate concolic operation as well as its approach to producing inputs. In the former case, PSE could sidestep the typical state explosion scalability issue by "jumping in" and beginning full symbolic execution deep within some program state. This also enabled cooperation between PSE and other input producers (for instance, GRR). In the latter case, PSE produced inputs at every symbolic fork. That is, the vast majority of inputs produced by PSE were produced before PSE observed the target program's termination. Frequent and rapid input production enabled deeper program exploration by the fuzzer.

KLEE. Early versions of Cyberdyne also integrated KLEE (https://klee.github.io), another open source symbolic executor. KLEE operates on LLVM IR (LLVM Intermediate Representation), a program representation used by the clang compiler. Typically, LLVM IR is produced by compiling C/C++ source code with clang. In the CGC, source code for target programs was not available. We overcame this challenge by using McSema (https://github.com/trailofbits/mcsema) to convert x86 binaries into LLVM IR.

Cyberdyne's KLEE bug-finding service was eventually deprecated because KLEE was challenging to understand and maintain and because its symbolic emulation was a "leaky abstraction." In the latter case, KLEE provides no isolation between its runtime (for example, memory and code used by its OS emulation functions) and that of the program being symbolically executed. This led to false positives (KLEE claiming inputs crashed the program) and false negatives (KLEE missing actual crashes).

Mixer

The mixer performs the crossover operation of a generic algorithm: it takes existing inputs with high code coverage and uses various heuristics to merge these inputs into a new candidate input. In most systems, the input-mixing stage occurs as a part of whole-file mutation prior to fuzzing, but because GRR fuzzes programs via DBT, Cyberdyne can provide more granular mixing. For example, the mixer can take two program traces and interleave an input system call from each trace to produce a new input. Or it can operate at the multiple-call or whole-file level, splicing two inputs in various ways.

While it sounds simple, the mixer was surprisingly effective. For example, imagine a program where you enter your name and then solve a maze. If you solve the maze, an overly long name triggers a bug at the high score screen. Using the mixer, a series of inputs that solves the maze would quickly propagate to other high-coverage inputs, like one that sets an overly long name. In effect, the "genes" of high-coverage inputs would mix together to form child inputs that result in higher code coverage than the parents'.

Minset

The minset service implements the fitness function of Cyberdyne's genetic algorithm. Fitness is measured

using code coverage: if an input induces the execution of previously unexecuted code, then that input is scored as interesting and is eventually fed to the other services. The goal of the minset service is to select a minimum set of program inputs that maximize code coverage.

Coverage-guided bug-finding systems have become commonplace in recent years. American Fuzzy Lop (AFL; http://lcamtuf.coredump.cx/afl), libFuzzer (http://llvm.org/docs/LibFuzzer.html), and SanitizerCoverage are all production-quality coverage-guided fuzzers. Some of these tools were employed by other teams in the CGC. For example, the Shellphish team used AFL directly, and the Codejitsu team used a modified version of AFL (https://github.com/mboehme/aflfast) that employed a new algorithm for prioritizing which inputs to fuzz.

Code coverage. There are many ways of measuring how much of a program is executed given a specific input. Cyberdyne uses an extended form of branch coverage as its code coverage metric. Before executing a branch instruction, Cyberdyne records a three-address tuple:

- the most recently executed branch instruction address,
- the about-to-execute branch instruction address, and
- the destination instruction address of the about-to-execute branch.

The most recently executed branch instruction can be arbitrarily far back in time, adding more sensitivity to this coverage metric.

Cyberdyne also counts indirect control flow instructions as branches (for instance, vtable-based method calls, jump tables implementing `switch` statements, and function returns). For example, function return instructions redirect control flow by jumping to an instruction address stored on the stack. Attackers can utilize stack-based buffer overflows to corrupt the stored return address and take control of a program's execution. Treating return instructions as branches allows Cyberdyne to detect return address changes as new coverage.

Lossy or lossless? What metrics count toward code coverage are just as important as how those metrics are recorded. A precise, or lossless, coverage recording adds any input that induces the execution of a previously unseen branch tuple into the minset. A lossy coverage recording has the potential to treat some inputs as not inducing new coverage even when it should.

In the CGC, we observed that importance of precision changed over time. The CGC was a competition, and points were scored when bugs were found, even if those bugs could not be converted into vulnerabilities. This incentivized the discovery of low-hanging fruit, that is, superficial bugs that manifest early in a program's execution, before the harder-to-find bugs are reached. Our experiments showed that most superficial bugs could be discovered within the first 30 minutes of fuzzing a target program, but only if the metric, or recording thereof, was extremely lossy. Hard-to-detect bugs, however, required precise coverage information recording.

We implemented adjustable precision control by taking inspiration from probabilistic data structures like Bloom filters. Cyberdyne recorded coverage by hashing each branch tuple and using that hash as an index into a file-backed bitmap. When a branch is executed, the instrumentation sets the bit associated with the hashed tuple. Low-precision recording was achieved by using a small bitmap file, or a hash function with a poor distribution, whereas high precision was achieved by using larger bitmap files and better hash functions.

Why did precision matter? The reason is a mix of resource allocation, scheduling, and brute-force effort. Shallow bugs can usually be discovered by brute force, that is, letting the fuzzer run its course of mutators given an input seed. Cyberdyne's fuzzer scheduler gives priority to "new" inputs and prioritized deterministic mutators (for example, flip the first bit of every byte) above nondeterministic ones (for instance, Radamsa). Starting a campaign with low-precision coverage recording helped keep the minset small, thereby bringing to bear more CPU resources per input in the minset.

> **After the CGC, the question was "what's next?" For us, the answer was retargeting the technology to audit the deployed programs and libraries that silently power the computing infrastructure.**

Deployment

We automated the provisioning and deployment of a Cyberdyne system. A multinode Cyberdyne system can be deployed on private or public clouds via a single command line. We have successfully run Cyberdyne on hardware ranging from a developer's laptop to hundreds of extra-large Amazon EC2 instances. This flexibility allows us to use Cyberdyne for something as large as the Cyber Grand Challenge or for something as small as a single-application code audit.

Using Cyberdyne

First, we effectively used Cyberdyne during the CGC. Cyberdyne identified the second-most amount of bugs

in the CGC's qualification challenge binaries. A faster, more accurate, and more efficient version of Cyberdyne was a key component of team Deep Red's bug-finding operations in the final event.

After the CGC, the question was "what's next?" For us, the answer was retargeting the technology to audit the deployed programs and libraries that silently power the computing infrastructure we take for granted.

We began an effort to use Cyberdyne to automatically identify bugs in Linux applications. We knew that Cyberdyne could find bugs in "real" software—the challenge binaries are most certainly real.[1] They exhibit complex behavior and vulnerabilities unknown to the authors. However, Cyberdyne was designed to find bugs only in software written for DECREE, an operating system purpose-built for the CGC. While DECREE is Linux based, it is very different from Linux.

Using Cyberdyne on Linux Applications

There are substantial design differences between DECREE and Linux. DECREE was designed to remove unnecessary complexity so competitors could focus on the science of program analysis instead of modeling quirks of operating system abstractions. Although DECREE is based on the Linux kernel, operating system features such as threads, sockets, files, and signals are not accessible to DECREE applications. Linux programs expect all these features to work, and more.

Rather than making the effort to port Cyberdyne to Linux, we realized that we could audit real programs quickly if we ported them to run in DECREE. The porting process requires little to no modification of the original program source. Most of the effort focused on modifying how the programs were built and simulating Linux functionality in DECREE.

Simulating Linux with DECREE. First, we created a libc implementation using only DECREE functionality. The core of libc was built from parts of open source libc implementations and from libc portions implemented in the challenge binaries. Where necessary, such as for file, thread, and process operations, we mocked the functionality. Most mock system calls either return a successful error code or error not implemented, but a few, such as `open` and `time`, required more complex modeling. File operations on key files such as `stdin`, `stderr`, and `stdout` must be accurately modeled to get input into the program. Time retrieval operations shouldn't make time go backward and should report a time close to the present day. Resource limits for `rlimit` should be realistic, as should `stat` results for files that are known to exist on a real Linux system.

Building programs to LLVM bitcode. DECREE programs are built using clang. Porting a Linux application to DECREE is a four-step process. First, we obtain the source code for the application and all of its dependencies. Second, we compile all relevant source files to LLVM bitcode modules. Third, we link all modules together into a single, unified LLVM bitcode module. Finally, we link this aggregate module to our custom libc implementation. The result is a single file that represents the program and all dependencies, down to the system call layer.

Emitting DECREE executables. Combining all dependencies in a single LLVM bitcode module enables substantial optimization and analysis opportunities. A typical program will use only a fraction of libc and dependent library functionality. Normal libraries are a package deal: there is no way to import only some functionality. Because our new build system merges dependencies at the LLVM bitcode layer, we can use LLVM's optimization passes to eliminate unused functionality and to flatten layers of indirection. Less code in the program means less code to analyze. Because Cyberdyne operates on binaries, we can gain more analysis advantages by emitting only instructions that are easy to reason about, especially for our binary symbolic executor PSE. For instance, we can avoid creating a binary with legacy FPU instructions or complex vector instructions like AVX.

Limitations. Our approach to porting applications to DECREE has limitations. First, this approach requires source code. While access to source code is not a problem for most Linux applications or code-auditing engagements, it removes certain classes of programs from analysis. Second, some applications will never be portable to DECREE: they may require graphical interaction, shared memory, or other features that can't be duplicated in DECREE. That shouldn't stop us from porting those components we'd most want to analyze—encoders and decoders, compression libraries, image processing libraries, parsers, and so on.

Next, let's explore how we applied Cyberdyne to perform the first paid automated security audit in history.

The First Paid Automated Security Audit

In August 2016, Cyberdyne audited zlib (https://github.com/madler/zlib) for the Mozilla Secure Open Source (SOS) Fund. To our knowledge, this is the first instance of a paid, automated security audit. Zlib is an open source compression library that is used in virtually every software package that requires compression or decompression. This very article was created and printed with multiple software packages that use zlib. Online readers

are also using zlib—the PDF viewer or web browser you are using relies on zlib for compression and decompression functionality.

Zlib has a relatively small code base that hides a lot of complexity. First, the code that runs on the machine may not exactly match the source, due to compiler optimizations. Some bugs may only occur occasionally due to use of undefined behavior. Others may be triggered only under extremely exceptional conditions. In a well-inspected code base such as zlib, the only bugs left might be too subtle for a human to find during a typical engagement.

Mozilla is a nonprofit and houses a variety of projects beneficial to the public. Our automated audit of zlib was thousands of dollars cheaper than an equivalent human-powered audit and provided measureable code coverage and generated inputs. The money saved can be put to good use supporting other Mozilla projects, and the generated artifacts can be easily reused for future automated or manual audits.

For this automated assessment, we paired Cyberdyne with TrustInSoft's verification software (https://trust-in-soft.com) to identify memory corruption vulnerabilities, create inputs that stress varying program paths, and identify code that may lead to bugs in the future.

Audit methodology. During the assessment, we focused on typical zlib usage and code related to compression and decompression functionality. Unrelated features were not audited, unless they were called by core compression or decompression routines.

Zlib is written in C and is designed to build for an extremely wide variety of platforms and compilers. Some code is built only for certain platforms (for instance, only big endian or only little endian). For the audit, we built zlib version 1.2.8 (the latest available at the time) using the clang compiler targeting the 32-bit Intel x86 instruction set.

Audit results. Cyberdyne is especially tuned for identifying memory safety violations (for example, buffer overflows, use-after-free errors, stack overflows, and heap overflows). Using Cyberdyne, we were unable to identify memory safety issues with the `compress`, `uncompress`, `gzread`, and `gzwrite` functions in zlib. We concluded that the assessed code was highly unlikely to harbor these types of bugs. The TrustInSoft analyzer identified uses of undefined behavior; the full report describes the details of these potential vulnerabilities.[2]

The full line and branch coverage results for the core zlib source files are shown in Table 1 (reproduced from the audit report). Cyberdyne automatically generated inputs to gather this coverage, given a program to exercise the correct functionality and minimal seed inputs to speed up the input synthesis process. Very high coverage for the location of previous zlib vulnerabilities, in the Huffman tree code (inftrees.c: 98.3 percent line coverage, 93.7 percent branch coverage, trees.c: 93.6 percent line coverage, 89.4 percent branch coverage), was a very welcome sign. The lone outlier was the file infback.c, which had 0 percent coverage. Its functionality was never invoked. Infback.c is an alternative to inflate.c and is only used when callback style I/O is preferred.

Table 1. Code coverage in zlib.

File	Line coverage (%)	Branch coverage (%)
adler32.c	67.20	61.80
compress.c	90.50	62.50
crc32.c	29.10	32.60
deflate.c	44.00	30.70
gzclose.c	80.00	75.00
gzlib.c	37.80	24.60
gzread.c	50.70	38.60
gzwrite.c	40.50	24.70
infback.c	0.00	0.00
inffast.c	83.80	75.70
inflate.c	75.30	65.90
inftrees.c	98.30	93.70
trees.c	93.60	89.40
uncompr.c	100.00	71.40
zutil.c	38.50	50.00

This coverage is a result of invoking the compression-related code both directly and via gzip functionality. This compares very favorably to the handcrafted unit tests that come with zlib, which generate 100 percent coverage for infback.c, inffast.c, and inftrees.c; 98.6 percent coverage for inflate.c; and almost zero coverage for anything else.

Cyberdyne's automated audit revealed no new memory safety violations in zlib. Far from a disappointment, the null result was expected. Zlib has been audited by humans and battle-tested by virtue of being deployed on almost every Internet-connected device. To date,

there have been no reported vulnerabilities in the tested version of zlib.

The audit was a learning experience for us on the benefits, challenges, and limitations of automated code audits. While the actual auditing is fully automated, the kind of library software Cyberdyne is best at analyzing requires a level of upfront manual effort.

Challenges of Automated Security Audits

Cyberdyne automates the process of generating program inputs, exercising new code paths, and identifying bugs. What Cyberdyne does not automate is porting Linux applications to DECREE or writing programs to exercise library functionality. Porting and exercising functionality still require careful analysis of how the software under test works and initial manual effort.

Building the target software. The biggest challenges in porting software to DECREE are identifying all build dependencies and modifying all the build and configuration systems to emit code for 32-bit x86 processors, disabling potentially problematic features (for example, handwritten vectorization, threading support, and so on), and using a customized version of clang that emits whole-program bitcode. Even for projects that build with GNU Autotools, simply changing the compiler is rarely sufficient. Some software won't build with clang. Other software requires custom configuration options or has a customized build step. Most build configurations are just different enough from one another that some human intervention in the initial build process is still required.

> **Automation will democratize access to security: low-cost, high-coverage testing will finally be within reach of individual developers and open source projects.**

Exercising program functionality. Exercising program functionality is the second big challenge of automated auditing. End-user applications have a single entry point (that is, `main`) but process several command line flags that affect the program. Shared libraries export a range of functionality via different exported symbols. How program or library functionality should be used is not defined at a machine level, but via documentation (which is sometimes lacking).

To illustrate, let's use zlib as a motivating example. The usual entry points into zlib are `compress` and `uncompress`, but zlib also offers gzip wrappers around the standard library file functions (for example, `gzopen`, `gzread`, `gzwrite` and so on), checksumming functionality, and more. To test the whole library, a program or multiple programs must be written to invoke these functions. In Cyberdyne parlance, the small programs that feed input to functionality under test are called drivers. Developing drivers to exercise all program functionality is a part of the initial manual test setup process required to use Cyberdyne to audit Linux software.

Driver development ranges from the trivial to the complex, depending on how many entry points the underlying software has. Applications that accept file inputs typically need a single trivial driver (for example, call the main function with `stdin` as the input file). Shared libraries require many carefully written drivers that demand a thorough understanding of the software under test.

Depending on the software under test, other factors may also come into play during driver development: How do you set bounds on inputs to software that accepts unbounded input lengths? How do you best mock system configuration files read by the software? What are realistic values for mocked resource limits?

Even though it sounds like creating drivers is a daunting task, automation comes to the rescue once again. Because the actual auditing is fully automated, driver development can be iterative. An initial driver just needs to feed input to some part of the software. That driver will then start the initial audit. As the initial audit runs, any newly developed drivers can be added to Cyberdyne to increase the breadth of tested code.

The Future of Automated Code Audits

Despite the challenges, automated code audits are the future of security testing. Human auditors are rare and expensive and simply cannot keep up with the volume of newly written code. Automation will democratize access to security: low-cost, high-coverage testing will finally be within reach of individual developers and open source projects. The revolution in security testing will dramatically improve the security posture of the Internet's core infrastructure.

The automated auditing revolution has already begun. Projects like Google's oss-fuzz project and Coverity Scan already enable continuous fuzzing and static analysis checking of open source applications. We envision a future where services like Cyberdyne integrate with common continuous integration and code-hosting environments to deliver quality, low-cost security audits. Initially, automated audits will be a complement to unit and integration tests—something that good software developers use to ensure they ship quality code.

As automated audit technologies mature, objective security metrics will slowly become meaningful and change the economics of software security for the better. Imagine if software came with verifiable guarantees of "audited with Cyberdyne to 99.99 percent branch coverage." The label wouldn't guarantee bug-free operation, because exploitable vulnerabilities may still exist in edge cases, but it would be meaningful way to compare software products. Once objective security measures influence software purchasing decisions, there will finally be an economic reward for writing software securely. The economic incentives will create a virtuous cycle of better analysis tools, higher-quality software, and a more secure world for all.

There are no magic numbers that make Cyberdyne find bugs. There are reasons why even just one single-threaded GRR process can perform a million mutate, execute, and code coverage measurement cycles per hour. The decisions that we made were guided by a performance-focused mindset, and backed up with measurements. Our approach is general and broadly applicable to other bug-finding systems. The approach provides a framework for tackling issues when scaling vulnerability discovery, for instance, sidestepping state explosion in symbolic executors.

The tools and concepts developed for the Cyber Grand Challenge can be used on practical software. Cyberdyne demonstrated this by performing the first paid automated security audit for the Mozilla SOS fund. The automated audit of zlib delivered higher confidence at a lower cost and faster timeframe than was possible with qualified human code auditors.

Automation represents a shift in the way that software security audits can be performed. It's a tremendous step toward securing the Internet's core infrastructure. Automation can deliver continuous, low-cost, and effective security analysis. Other projects have already adopted this model: Google's oss-fuzz project enables continuous fuzzing of open source applications. We envision a similar future for Cyberdyne: a service that works with popular code-hosting and continuous integration environments and delivers quality, low-cost, continuous security audits. As automated tools improve, we believe they can finally upend the economics of software development to reward safety instead of features by providing objective, comparable security metrics across disparate software packages. ■

As automated tools improve, we believe they can finally upend the economics of software development to reward safety instead of features by providing objective, comparable security metrics.

References

1. "Your Tool Works Better Than Mine? Prove It," Trail of Bits blog, 1 Aug. 2016; https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it.
2. A. Dinaburg and P. Cuoq, *Zlib: Automated Security Assessment*, Trail of Bits, 30 Sept. 2016; https://github.com/trailofbits/public-reports/blob/master/Zlib-AutomatedSecurityAssessment.pdf.

Peter Goodman is a senior security engineer at Trail of Bits. He is an expert at designing and implementing binary translation and instrumentation systems. He holds an MS from the University of Toronto, where his research focus was operating system kernel instrumentation. In a past life, he was a competitive ski racer. Contact at peter@trailofbits.com.

Artem Dinaburg is a principal security engineer at Trail of Bits. His current interests include automated program analysis and creating program analysis tools usable by software developers. He holds a bachelor's in computer science from Penn State and a master's in computer science from Georgia Tech. He has previously spoken at academic and industry computer security conferences such as ACM CCS, INFILTRATE, Blackhat, and ReCON. Contact at artem@trailofbits.com.

Privacy-Aware Restricted Areas for Unmanned Aerial Systems

Peter Blank, Sabrina Kirrane, and Sarah Spiekermann | Vienna University of Economics and Business

Although drones are receiving a lot of attention, the protection of citizen privacy is still an open issue. To this end, we demonstrate how basic principles of information privacy could be integrated with existing infrastructure to build a framework for privacy-aware unmanned aerial system (UAS) dispatch considering restricted areas.

Unmanned aerial vehicles (UAVs), otherwise known as "drones," are flying platforms that are controlled remotely. Recently, drones, which come in many shapes and sizes, have experienced increasing attention due to expanding capabilities, such as longer flight durations and interesting application areas (for instance, face recognition or tracking). UAVs combined with analytical systems form unmanned aerial systems (UASs), which have already shown great potential especially in governmental applications, such as law enforcement and rescue service domains. An early investigation into this potential was conducted in 1999, when Murphy and Cycon proposed the use of mini-UAVs in law enforcement surveillance applications.[1] A subsequent study in 2007 by the European Commission indicated high potential usage of UASs not only by law enforcement, but also by border security and the coast guard.[2] Already police forces have started testing and deploying UASs in practice. For example, UASs have been used for crime scene or accident photography[3] and to locate (injured) persons.[4] Considering the ongoing advancement of UAS technologies, UASs have the potential to become an important support in a variety of public and private sector activities.

That said, drones can cause privacy harms as they can potentially invade people's private space, and accidentally expose them by processing personal data against their will. Additionally, privacy violations can occur through the unsuspecting collection of information concerning random citizens without any purpose, simply due to constant video recording while flying. For example, Finn and Wright[5] show that UASs can be used to monitor large crowds for border patrol or crime prevention. Police-operated UAVs may frequently cross private property on their way to an operational area, for instance, when flying to an accident or simply monitoring an area. For citizens living in the approach corridor, near the landing and starting places of UASs, or frequently passed routes, this can be disturbing because of the close proximity and noise, the frequency, or both. With a UAS sensor system capturing citizens' property, a citizen can be recorded, identified, or recognized on his or her property even though the reason for dispatch is another subject or target.

From a societal perspective, it is necessary to assess privacy-related considerations, such as the impact constant flying and filming of property in the surrounding area has on individuals. From a legal perspective, it is necessary to get consent from individuals before they are recorded and also to provide transparency with respect to the data that is captured and the type of processing. Although this could prove burdensome for

 Copublished by the IEEE Computer and Reliability Societies

UAS operators, compliance is imperative to gain broad acceptance of the technology by society. One way to avoid unnecessary or unintended privacy violations is to check the flight path for potential privacy violations a priori. Depending on the reason for UAV deployment, for instance, in the case of maintenance or routine flights, flying over private property can be avoided by rerouting the UAV.

Despite these obvious privacy issues, recent research efforts focus primarily on optimizing the UAV platform, such as UAS routing in larger groups (swarms), their sensor capabilities, or software algorithms. Most of the emphasis is put on the technical advancement of potentially privacy-invasive activities, such as monitoring of crowds or airborne surveillance and tracking. However, the legal cases around the deployment of drones in the US already show that privacy-aware UAS deployment is of crucial importance for UAS usage. This article aims to address this gap by examining privacy challenges associated with tactical UASs. We propose a framework that deals with privacy-aware UAS deployments by granting citizens some degree of control over the UAS flight paths. The proposed dynamic UAS routing framework can be used to ensure that drones do not fly over private property.

Privacy and Drone Deployment—The Status Quo

When it comes to UASs and privacy-related research in the US, to date, the focus has been on the legal implications. McBride highlights that law enforcement agencies are pushing for (small) UAVs in tactical operations supporting ground police units.[6] He further discusses three famous Supreme Court decisions regarding aerial surveillance by police forces. In the first of these three cases, *California v. Ciraolo*, the police were informed of illegal activities on a private property. The police forces subsequently deployed an airplane to fly over a private property, which was protected by fences, and took photos of growing marijuana. The Supreme Court dismissed the case as the police forces violated the privacy rights of the suspect. In *Dow Chemical Co. v. United States*, the Supreme Court ruled that privacy privileges in regard to aerial photography do not apply in the context of industrial facilities spanning over 2,000 acres. The Supreme Court decided that commercial property is subject to the "open fields doctrine" as opposed to the "curtilage doctrine." The third case investigated was *Florida v. Riley*, where police observed a partially overgrown greenhouse, located in a backyard. The observation, which was carried out from a helicopter, was judged to be a "search" for which a warrant would have been required.[6] Together, these three cases have had a major impact on aviation-based surveillance under the Fourth Amendment in the US, as they defined aerial surveillance as a "search" when a considerable expectation of privacy is breached. However, such an expectation is not applicable for industrial facilities, but rather protects private property under the curtilage doctrine. Following the privacy discussion about UASs, Calo[7] expects UAVs to become a privacy catalyst. He predicts that UAS-related privacy concerns will "gain serious traction among courts, regulators and the general public."[7]

When it comes to UAS privacy in Europe, a number of laws regarding the design of privacy-aware drones need to be considered; mainly the EU data protection directive (95/46/EC), the EU Regulation on the protection of processing by community bodies and the free movement of data (2001/45/EC), the General Data Protection Regulation (enforcement to start in May 2018), and the forthcoming Data Protection Directive (DPD) for public authorities. The upcoming EU regulation specifies, among others, the impact of compliance breaches, which are discussed later in the article. However, as community law is usually not applicable for law enforcement, when it comes to public authorities, the DPD is the most relevant. Although there has been very little research on privacy for UASs, Cavoukian[8] proposes several comprehensive privacy-by-design principles. At the most basic level, UAS privacy should be proactive and preventive, it should be the default setting, and usable without adaptions. It must be an integral part of the solution, embedded into the system from the very beginning. It needs to be fully functional in coherence with other relevant principles, such as security. Additionally, visibility and transparency with respect to processing are required to gain broader acceptance from the public and various stakeholders. Essentially, respecting privacy requires UASs to offer user-friendly options, such that, for example, strong privacy defaults are set and adequate warnings of UASs gathering data are set up in the relevant regions. Also, the lifetime protection of the proposed privacy mechanisms should be guaranteed via regular privacy impact assessments that allow privacy principles to be translated into required actions, which can be addressed by privacy-enhancing

"Visibility and transparency with respect to processing are required to gain broader acceptance from the public and various stakeholders."

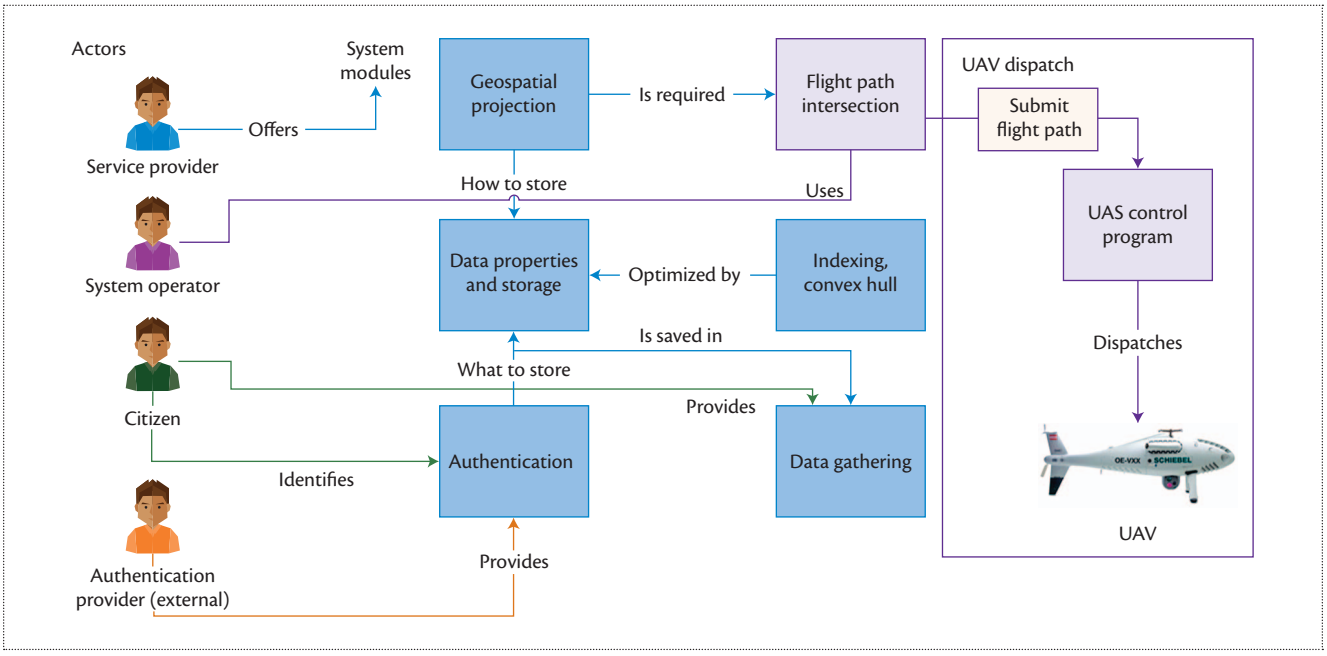


Figure 1. Software system architecture.

technologies.[5,9] Further principles for personally identifiable information in information privacy are discussed in a Council of European Union document,[10] which specifies that organizations must process data only collected for specific, explicit, and legitimate purposes that are relevant, adequate, and limited to this purpose; processed lawfully and fairly; and transparent for the data subject. Also, data must be accurate and kept up to date with reasonable effort, kept in a form that does not permit identification, and be processed under the responsibility and liability of a controller that has to show compliance with the regulation.

Technical Advances to Protect Citizens' Privacy Rights through UAS Routing

Against the background of the legal situation, it is important to see where we stand in terms of privacy by design for UASs: What technical research and development could be used to address some of the legal thinking? Can we technically ensure that police forces don't violate the private rights of individuals by crossing their territory, potentially filming? In Europe, we expect that citizens' consent to UAS data collection over their private property could become an issue, as well as the desire of citizens not to be disturbed by UAVs—that is, their right to be let alone. One crucial enabler necessary in order to respect citizen preferences is to equip UASs with intelligent routing technology. Smart routing technology would allow UAVs to routinely avoid areas that are marked as private.

To date, a number of authors have tackled and industry has proposed routing algorithms for UASs. DJI, a UAV manufacturing industry company, developed a static approach for restricted area recognition in which property coordinates are compiled into the UAS software. By forcing GPS-equipped UASs to stop at the border of a restricted property, the software allows the UAS to recognize restricted areas without requiring Internet access. This approach is intended for small UAVs and is used to block access to specific properties, such as airports or military areas.

The UAS will stop at the very border of the restricted area (in both height and planar coordinates) and wait for new instructions while hovering. If the UAS runs out of energy, it tries to perform a safe emergency landing at its current position. Another complementary industry solution is offered by the website NoFlyZones.org. Participating UAV manufacturers are able to consult a database that contains names and addresses of property owners, and integrates the relevant restriction into the UAV software, by looking for coordinates that are associated with the addresses.

One of the limitations of existing systems is the fact that property restrictions are compiled into the UAS software, meaning it is difficult to make changes. In addition, it is not possible to specify permissions for specific contexts; for example, in an emergency situation, a privacy infringement may be acceptable. The framework proposed in this article intends to solve these problems by catering for real-time querying of context-specific

permissions and by enabling citizens to amend these permissions at any time. As the software itself does not need to be updated, citizens are given a greater degree of control over their privacy preferences.

A Framework for Privacy-Friendly UAS Routing

The proposed privacy framework, which is depicted in Figure 1, distinguishes between four types of actors: system operators, service providers, citizens, and authentication service providers. When considering police forces using UASs, the system operator can be a police officer in a UAS control center, and the service provider can be a police organization or a ministry. Any citizen holding a legal property title may use the system to set their privacy preferences. It is envisaged that the authentication service providers are trusted e-identity providers. These actors interact with six different modules in our privacy framework: First, the property coordinates must be represented using a specific geospatial projection and associated with certain attributes, for instance, specific permissions for flying over a property. To enter data, citizens need to identify themselves via an authentication infrastructure, which is offered by an (external) authentication provider. After authentication, the citizens can enter details about their private properties using a web interface that is offered by the service provider. Based on the data input, a checking entity is required to confirm the correctness of the request. After the correctness check, convex hulls can be calculated to increase the efficiency of the calculation of the flight path. The system operator can select and request the flight path calculation from the system. If there is no intersection between flight path and restricted areas, the flight path can be submitted to a UAS control program. Finally, the UAS control program handles the communication to the UAV, allowing it to be dispatched according to the flight coordinates chosen. This section provides a detailed description of each of the six system modules.

Representing Geospatial Data Using a Digital Map

The selection and assignment of a geospatial projection, which is a representation of the Earth's surface, is of utmost importance for the accuracy of the UAS's flight path selection. The flight path and the coordinates used for storing restricted areas are displayed in the chosen projection on a digital map. To date, there is no perfect representation of an oblate spheroid; therefore, projections differ significantly and are even prone to distortions. The most common projections are either cylindrical (for instance, Mercator projection), planar (for instance, unprojected latitude and longitude), or conic (for instance, Lambert conformal conic). Figure 2 depicts an example of those differences by showing three projections of the United States in red (Mercator), blue (Lambert conformal conic), and green (unprojected latitude and longitude). The distortion of projections can also be assessed by consulting Google Maps, which uses the Google Web Mercator WGS84 (EPSG:3857) projection. By comparing Greenland to Australia on Google Maps, Greenland appears much larger than Australia when in fact it is only approximately 28.16 percent of its size. Once the geospatial projection is assigned, geometric data types such as GEOM can be used to store coordinates associated with private properties as polygons. Therefore, it is necessary either to use a standard projection, for example, the Mercator projection, that is compatible with the particular GPS sensor of the UASs or to convert between the sensor data and another projection on demand. Spatial database extensions such as PostGIS allow for the conversion between or storage of coordinates associated with different projections.

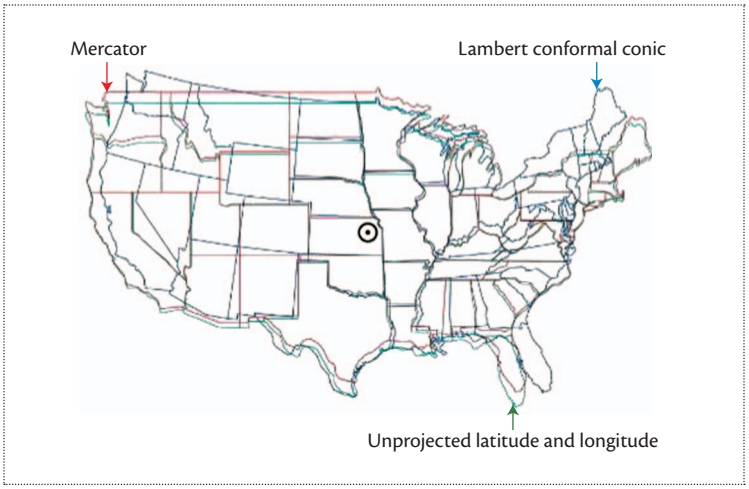


Figure 2. Different projections and their deviations: three projections of the United States in red (Mercator), blue (Lambert conformal conic), and green (unprojected latitude and longitude). (Adapted for better readability from Peter H. Dana, "The Geographer's Craft Product," 1997. Used with permission.)

Efficient Storage and Querying of Property Data

The data properties and storage module caters for the efficient storage and querying of property data that is gathered via a web interface. As property databases may need to hold a vast amount of data and real-time processing may be necessary for privacy considering police operations, the efficiency of the database is of utmost importance. For instance, if one is considering Germany, the private-house-ownership rate is estimated at 48 percent (approximately 19 million).[11] If only

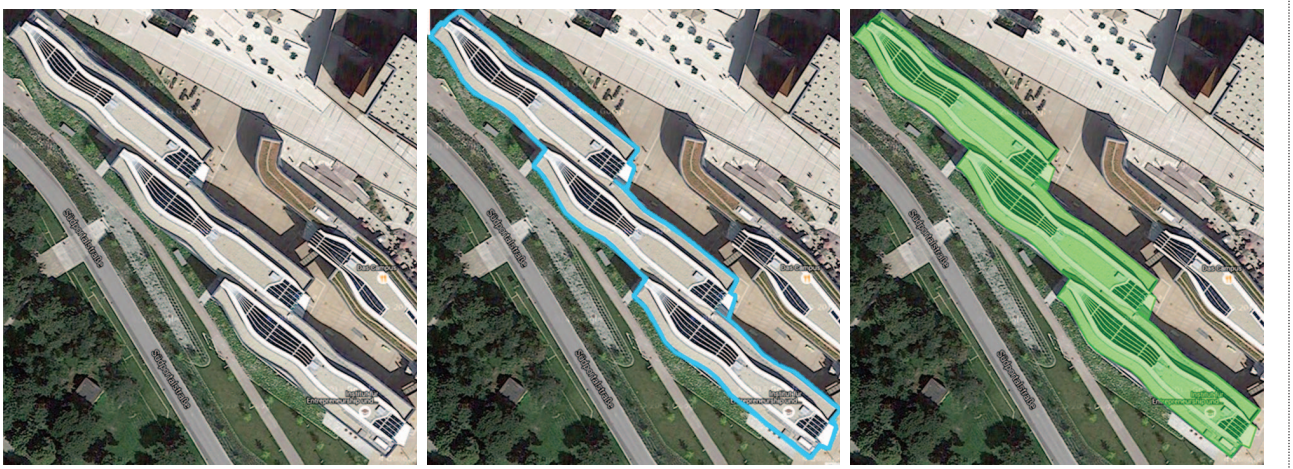


Figure 3. Polygon registration via map interface: (a) satellite image by Google Maps, (b) selecting property borders (blue line), and (c) registered polygon (solid space—green).

8 percent of these house owners define a privacy preference and each property has several border points (let's assume five), one would find 7.6 million entries for Germany alone. Furthermore, for each property, there can be a set of optional contextual permissions stored, for example, that a police UAS may fly over a property only in emergencies, but not for routine flights. Although the search space increases with the number of additional properties, scalability can be achieved by indexing the data based on attributes that are commonly used for querying, such as the number of coordinates, regions, and political areas.

In some European countries like Germany, Austria, Great Britain, and France, there is no complete, direct mapping between the land register or cadastral map and any of the available projections. In the United States, the land registration is a matter of state regulation such that only public lands of the US are centrally mapped by the Bureau of Land Management. China currently dictates that all land ownership and leaseholds are recorded in an official register; however, there is no general procedure on how to register a property. To achieve a mapping between a projection and the registers of restricted areas, there are two different approaches to be considered. First, the public authorities could include the EPSG projection coordinates in their current registers and introduce a converting schema or update their existing registers to a digital format. However, it appears to be rather complex to establish a general mapping into an EPSG coordinate–based digital format. Second, it is possible to allow user-generated input by choosing the coordinates via a digital map. The latter may appear to be less costly in the first place, but significant effort may be needed to check the correctness of the user-entered data. Such an approach would require a checking entity to gather information about the user's identity—whether the user is the property owner—and to verify the property borders in the chosen projection. Without the interconnection of existing data sources, such a validation of property claims cannot be conducted in an automated way.

Citizen Authentication

When it comes to authentication, where available, an existing digital (e-) identity infrastructure for citizens can be used to verify individual users' identities via electronic signatures. In Europe, an eIDAS Regulation governs electronic identification services, trust services, electronic signatures, and seals.[12]

Notably, several European countries, including Austria, Belgium, and Estonia, among others, have already set up infrastructures that support e-identities. By using the existing e-identity infrastructure for authentication, users can register certain polygons as their private property. A checking entity can subsequently use the public land register to check the claim. Users may specify different kinds of permissions. Possible options could include: "permit flights," "prohibit flights," or "permit flight for emergencies operations only." Permissions may be changed at any time by the user.

Recording Property Coordinates and Privacy Preferences

Given the unavailability of relevant data about private properties and their coordinates in the form of projections, we chose the second data-gathering approach, which involves user input. A digital map, which is presented to the user (see Figure 3a), allows citizens to

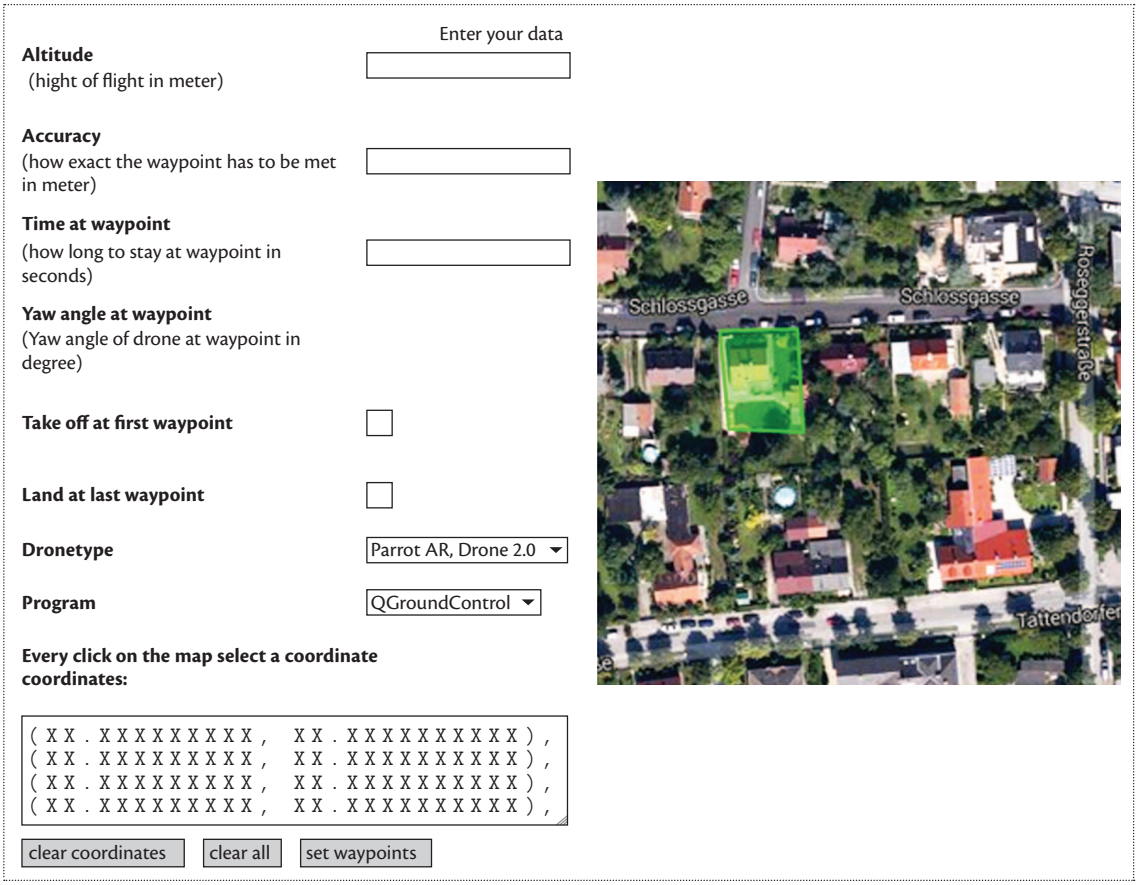
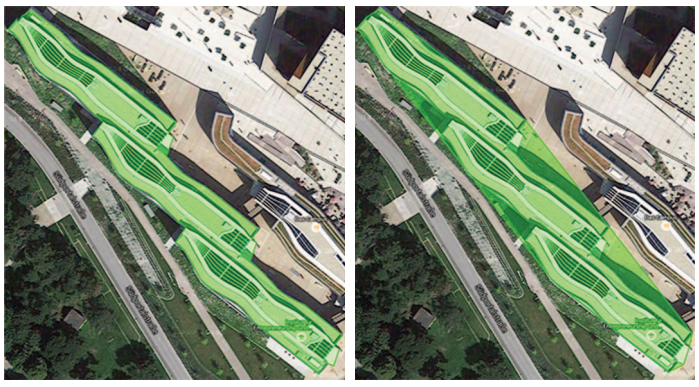


Figure 4. Operator interface for UAS dispatch.

enter their information (for instance, contact details, restricted areas, and permissions). By clicking on the borders of the relevant property, a polygon is generated on the map and its coordinates are fetched from the map, as depicted in Figure 3b. Privacy preferences are entered via a webform similar to the operator interface depicted in Figure 4. Once the user has confirmed that the private property is completely and correctly covered, the polygons' border points are stored as coordinates in the database. User input may be edited, declined, or accepted by a checking entity, for example, the service provider. This entity has to take care that users' property claims are correct and the property borders are defined accurately, for instance, based on the prior mentioned land register and cadastral maps. After the correctness has been verified, an overlay, which is visualized in Figure 3c, is used to depict a restricted area and further serves as input by the flight-routing algorithm.

For public or commercial users, the treatment of areas according to area classes can be useful. Special areas such as airports, military facilities, embassies, emergency areas, or venues on specific dates and times may require extended restricted fly areas for UAVs, larger than their actual property border. Such properties may be assigned a different class and color than "normal" private property. The calculation of a buffer zone around such areas can be generated, for instance, a restricted fly zone of 5 km for military facilities and 3 km for airports or emergency services. Depending on the country, different guidelines for the minimum distance of the buffer zone may apply. For instance, the US demands that UAV pilots that come within a five-mile radius of an airport contact the airport authorities[13] to inform them about the flight. Additionally, buffer zones that cater for safety-related issues, such as accidents, fire bursts, or shootings, could be set up dynamically. For example, UASs belonging to private persons or media professionals may not be allowed to enter the wider area, whereas UASs operated by police, ambulance, or firefighter forces may be permitted access. To calculate such a buffer, two general approaches can be taken: first, each coordinate of an area is enlarged by a certain distance, or second, a centroid coordinate is chosen and a circular buffer is calculated based on the required distance from the centroid. The first solution



(a) (b)

Figure 5. Convex hull of a property: (a) registered polygon and (b) convex hull.

results in an exact, extended geometry of the object that can be based on the distance calculation. Using a buffer that extends the coordinates by a specified distance may be preferable if high accuracy is demanded. Choosing the second option results in a circular shape around a central point, which usually requires less calculation effort and data transmission. However, it requires the additional effort to calculate a centroid.

Reducing the Number of Property Coordinates

A convex hull of a property is the smallest convex polygon that encloses the property. By using convex hulls, the coordinates that need to be stored for each property can be decreased significantly, however the knock-on effect is that the accuracy of the restricted area is reduced (see Figure 5). A convex hull may never be smaller than the actual polygon it is calculated for, but can hold an equal or smaller amount of points (coordinates). With fewer coordinates, flight path processing becomes more efficient as fewer intersection testings between the flight path and the restricted areas are required. For calculating convex hulls, we used the Quickhull algorithm, which is based on the Quicksort algorithm.

Flight Path Selection and Calculation

A police operator uses the operator interface to select and calculate a flight path. The interface allows waypoints (coordinates) to be selected and information about the following to be entered:

- altitude,
- required accuracy,
- stay time at waypoint,
- yaw angle at waypoint,
- takeoff at first waypoint,
- land at last waypoint,
- selectable UAV type, and
- selectable UAV control program.

The interface is depicted in Figure 4. The flight path evaluation is based on the intersection between the path and either the coordinates of the convex hull or the original coordinates. By intersecting each pair of coordinates from the polygon with each pair of coordinates from the flight path, it is possible to detect infringements. Once an intersection is found, the use of the flight path is prohibited by deactivating the export of coordinates to the control program. In the case of an infringement, the operator has to select a new route and test the route again.

Another approach would be to deactivate the sensor recording when flying over a private property. This could in principle be based on the framework above, by changing the requirement from "avoid flight paths" to "deactivate the sensors." Notably, the noise and visibility of the UASs would not be solved by extending the framework to cater for sensor deactivation. Additionally, sensor systems like cameras can record an area even if they are not above or in close proximity to it. Thus, a simple deactivation of sensors when flying over a property is insufficient. For a privacy-aware system, a more complex calculation that considers contextual information from the sensor system would be required. Alternatively, one could edit the sensors recording in the live stream from a camera, such that only those properties without a restriction are recorded. This can for instance be done by secure visual object coding, pixelating the recorded data stream.[14]

Prototypical Implementation

To verify the effectiveness of the proposed framework, we developed a web-based prototype that uses software known as QGroundControl (QGC) to interact with a Parrot AR.Drone 2.0. A Microsoft SQL Server database with spatial functions was used for data storage and querying. The operator and citizen interface were developed in HTML5, CSS, JavaScript, and PHP, and used the Google Maps API. The flight path intersection testing was implemented in TSQL, while PHP was used for other server-side processing, such as calculating convex hulls with the Quickhull algorithm when setting new restricted areas. Server-side processing is required to use thin clients, such as mobile phones. JavaScript, together with the Google Maps API, enables citizens to select their property and operators to view restricted properties, as shown before in Figure 3. A screenshot of the user interface is presented in Figure 4. Although the responsiveness of the map decreases with the number of restricted areas to be displayed, this limitation can be

mitigated by the just-in-time loading of properties that are within the borders of the current map.

The police operator interface caters for the configuration of the flight path attributes outlined in the preceding section. These attributes are based on the capabilities of the QGC software, which was used for testing the software. Whereas altitude, accuracy, takeoff, and landing are mandatory attributes, the stay time and yaw angle can be left empty. The UAV type and control program selection attributes provide support for different control programs and give metrics about the flight duration, based on the average speed of the UAV. QGC allows routing information to be imported or exported and handles the communication with the UAV via the MAVLink protocol. Moreover, distance and flight time calculations are performed via PHP and displayed on a flight-checking interface. To distinguish between different property categories or permissions, different color schemas were introduced. Green denotes private property, orange airports, and red military zones. A blue line is used to depict the flight path in the Google Map. The buffer around specific areas have not been implemented, but could be introduced by using a mechanism comparable to the distance calculation that was implemented as part of the flight metrics.

Evaluation of the Prototype

The effectiveness of the proposed system is evaluated from both a usability and scalability perspective.

Evaluating the Usability of the User Interface

In the first experiment, 13 people were asked in isolation (that is, the participants were not able to talk to one another after the sessions) to use the software and apply a think-aloud technique. The participants were asked to create at least one restricted area by themselves and to select several flight paths, whereby one or more flight paths intersect with their restricted area. Participants were asked to verbalize their thoughts immediately when interacting with the system.[15] Based on the feedback received, we refined the layout and added a more precise instruction manual. While the map itself was intuitively usable and demanded no further work, changes were made to achieve a more usable menu structure. In addition, buttons, an introductory tutorial describing how to use the software, and the export functionality were developed. Also, as some participants asked for the duration, distance, and whether the selected UAS type could even make it, more information was offered once the route was selected, for instance, the flight distance and the minimum time required to reach the destination.

In the second experiment, eight participants were asked to evaluate the user interface. To better resemble a real-world application, basic authentication functionality was introduced and dummy accounts, tailored to the participants' names, were constructed and tested. Again, a think-aloud technique was employed; however, on this occasion, only minor changes were suggested by participants. Most participants intuitively constructed restricted areas for either their own or their relative's buildings. The terminology used, for instance, UAS or UAV, was updated to make the software more accessible. The coloring used on top of the Google Map was adapted to cater for easier selection and better visibility, as suggested by several participants.

After incorporating the aforementioned changes, in the final experiment, a real-world application evaluation was conducted. For this experiment, two small restricted areas were assigned to a field in lower Austria. One restricted area was used with its convex hull, the other without. Six participants selected several flight paths that either went around or through the restricted areas. If an intersection with a restricted area was found, the coordinate export was disabled, making a start of the UAS with QGC impossible. The participants managed to successfully deploy the UAS via QGC, while selecting a flight path without restricted areas, with no further instructions or help.

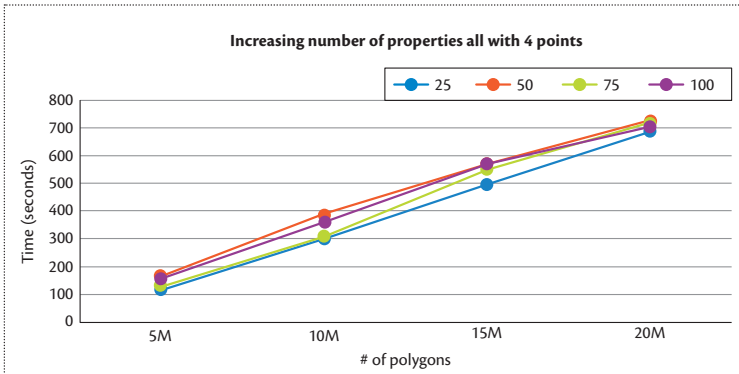


Figure 6. Query performance over increasing datasets.

Examining the Performance and Scalability of the System

To test the feasibility of the approach in terms of performance and scalability, we simulated 20 million polygons. This is slightly more than the aforementioned 19 million private households in Germany.[11] In the experiment described above, all participants were asked to create their own restricted areas.

Those areas consisted of several points. However, after applying the convex hull algorithm, we found that the majority of polygons consisted of only four points. Thus, the evaluation was conducted over increasing

datasets containing between 5 and 20 million polygons (5M, 10M, 15M, and 20M) with a relatively small number of points per polygon (that is, 4, 6, 8, and 10). To assess the impact of increasingly complex flight paths, we also tested for increasing flight path complexity, whereby a flight path had several different points that needed to be checked (that is, 25, 50, 75, and 100).

Figure 6 shows querying scales linearly with the number of points that need to be checked. Polygons with 6, 8, and 10 points exhibited similar behavior. Likewise, the disk space requirement to store the 5M, 10M, 15M, and 20M polygons scales linearly (that is, 563,408, 1,126,776, 1,690,152, and 2,253,536 kilobytes respectively). To improve the efficiency of the flight path checking, it is possible to generate indexes based on commonly queried attributes, for example, the range of coordinates, regions, and political areas. By reducing the search space, for example, to 5 million polygons, we find that the flight path testing takes on average 143 seconds for a flight path with 4 points. At 10 million polygons in the same setting, we find a duration of 338 seconds. However, it is worth noting that in a realistic setting, the number of properties along a route that need to be checked would be much less than 5 million.

The permission to operate UASs on a large scale will be heavily dependent on the operators' ability to comply with both data protection and aviation regulations.

Social and Legal Implications

If privacy-aware solutions are not introduced, not only will social acceptance be affected but the legal implications can be drastic. In Europe, failing to comply with the upcoming data protection regulations, such as processing data without sufficient legal basis like consent, may result in fines of up to 2 percent of annual turnover or up to 1 million, whichever is higher.[10] In the US, fines have been traditionally higher, but data protection regulations less strict, which could lead to comparable effects. Yet, not only are fines relevant for operators, but the permission to operate UASs on a large scale will be heavily dependent on the operators' ability to comply with both data protection and aviation regulations. However, UAS operators face a lot of uncertainty with the upcoming and existing regulations.[6] Additionally, a lack of sophisticated privacy technologies for UAS is making compliant UAS operating even more difficult for both commercial and private operators.

Frameworks such as the one proposed in this article are a first step toward greater privacy awareness and legal compliance in the field of UAS operations. However, it is worth nothing that while legal compliance can be imposed more easily on commercial operators by regular controls and checks, in the case of private operators, it may be difficult to enforce. For example, compliance could be circumvented by UAV operators that build their own drones or disable the privacy protection mechanisms of commercial drones. Although the risk of getting caught is negligible, we argue that high penalties should be put in place to discourage such behavior.

The proposed framework is a steppingstone toward more socially acceptable and legally compliant UASs. Guided by the privacy-by-design principles proposed by Cavoukian,[8] we describe how the proposed framework can be used to enable a proactive and preventive approach to privacy whereby respecting privacy is achieved by obtaining consent for flying over private property. In the proposed framework, privacy is the default setting that is embedded into the system from the very beginning. Although the system is fully functional from a consent perspective, further discussion is needed to determine what form of visibility and transparency is appropriate in such a setting. Also, when it comes to the lifetime protection of the proposed privacy mechanisms, further research is needed, especially in the context of the recording capability that is built into many UAVs.

One of the limitations of the existing system is the fact that even if the UAV does not fly over a certain property, a sensor, for instance, a camera lens, may be able to take records of it. Depending on factors such as altitude, camera angle, and yaw angle, records can be taken from a significant distance, which is not even close to a restricted area. Furthermore, our approach is limited to those citizens holding a legal property title. Nevertheless, when it comes to privacy-by-design solutions for UASs deployment, the proposed solution is a fundamental building block, on top of which other applications, such as calculating which parts of an image have to be pixelated, can be built. ■

References

1. D.W. Murphy and J. Cycon, "Applications for Mini VTOL UAV for Law Enforcement," *Enabling Technologies for Law Enforcement and Security*, International Society for Optics and Photonics, 1999, pp. 35–43.
2. "Study Analysing the Current Activities in the Field of UAVs," European Commission, Enterprise and Industry Directorate-General, ENTR/2007/065, 2007.

3. E. Pilkington, "'We See Ourselves as the Vanguard': The Police Force Using Drones to Fight Crime," *The Guardian*, 2014.
4. C. Franzen, "Canadian Mounties Claim First Person's Life Saved by a Police Drone," The Verge, 10 May 2013; www.theverge.com/2013/5/10/4318770/canada-draganflyer-drone-claims-first-life-saved-search-rescue.
5. R.L. Finn and D. Wright, "Unmanned Aircraft Systems: Surveillance, Ethics and Privacy in Civil Applications," *Computer Law & Security Review*, vol. 28, no. 2, 2012, pp. 184–194.
6. P. McBride, "Beyond Orwell: The Application of Unmanned Aircraft Systems in Domestic Surveillance Operations," *J. Air L. & Com.*, vol. 74, 2009, p. 627.
7. R. Calo, "The Drone as Privacy Catalyst," *Stanford Law Review Online*, vol. 64, 2011, pp. 29–33.
8. A. Cavoukian, *Privacy and Drones: Unmanned Aerial Vehicles*, Information and Privacy Commissioner of Ontario, Canada Ontario, Canada, 2012.
9. M.C. Oetzel and S. Spiekermann, "A Systematic Methodology for Privacy Impact Assessments: A Design Science Approach," *European Journal of Information Systems*, vol. 23, no. 2, 2014, pp. 126–150.
10. "Proposal for a Regulation of the European Parliament and of the Council on the Protection of Individuals with Regard to the Processing of Personal Data and on the Free Movement of Such Data (General Data Protection Regulation)," Council of European Union, 2012; http://eur-lex.europa.eu/legal-content/en/ALL/?uri=CELEX:52012PC0011.
11. "Homeownership Rate in Selected European Countries in 2014," Eurostat, 2014; www.statista.com/statistics/246355/home-ownership-rate-in-europe.
12. "Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on Electronic Identification and Trust Services for Electronic Transactions in the Internal Market and Repealing Directive 1999/93/EC," European Commission and Frost & Sullivan, Official Journal of the European Union, 28 Aug. 2014.
13. "FAA Modernization and Reform Act of 2012," 2012, Public Law No: 112-95, 112th Congress.
14. K. Martin and K.N. Plataniotis, "Privacy Protected Surveillance Using Secure Visual Object Coding," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 18, no. 8, 2008, pp. 1152–1162.
15. J. Nielsen, T. Clemmensen, and C. Yssing, "Getting Access to What Goes on in People's Heads?: Reflections on the Think-Aloud Technique," *Proceedings of the Second Nordic Conference on Human-Computer Interaction*, ACM, 2002, pp. 101–110.

Peter Blank is a process and data analytics professional at PwC Switzerland. Before that, he was a research assistant at the Vienna University of Economics and Business, where he primarily focused on privacy-related drone questions. His main fields of interest are privacy for emerging technologies, process mining, and data analysis in spatial applications. Contact him at Peter.Blank@ch.pwc.com.

Sabrina Kirrane joined the Vienna University of Economics and Business as a postdoctoral researcher in September 2015. Prior to this, she was a researcher at the Insight Centre for Data Analytics, Ireland. Her PhD focused on the problem of access control for the web of data. Before that, she spent several years working in industry on topics around data integration and security, such as system security requirements and their implementation in an application service provider environment. Kirrane's research focuses on the privacy issues that can result from interlinked machine-readable data. Kirrane is a guest editor for the *Journal of Web Semantics* special issue on Security, Privacy and Policy for the Semantic Web and Linked Data and besides others organizer of the PrivOn workshop series on Society, Privacy and the Semantic Web—Policy and Technology. Contact her at Sabrina.Kirrane@wu.ac.at.

Sarah Spiekermann is a professor for business informatics at Vienna University of Economics and Business and chairs the IMIS. Her main areas of expertise are electronic privacy, RFID, personalization/CRM, attention and interruption management (notification platforms), and context adaptivity. She advises the EU Commission in the area of privacy for RFID, served as a reviewer of RFID FP7 projects SMART and BRIDGE, coauthored and negotiated the PIA-Framework for RFID (signed in April 2011 by the EU Commission), and developed the PIA guidelines for privacy-friendly RFID systems for the German Federal Institute of Information Security (BSI) (published in November 2011). Contact her at Sarah.Spiekermann@wu.ac.at.

Cyber Deception: Overview and the Road Ahead

Cliff Wang | North Carolina State University and US Army Research Office
Zhuo Lu | University of South Florida

New research interests in adaptive cyber defense motivate us to provide a high-level overview of cyber deception. We analyze potential strategies of cyber deception and its unique aspects, discuss the research challenges of creating effective cyber deception–based techniques, and identify future research directions.

The notion of a cyber kill chain[1] was introduced to outline the stages of cyber adversaries, from early reconnaissance to the actual attack. Instead of engaging with adversaries in the early steps of the cyber kill chain (the reconnaissance phase), current cyber defense practices mostly focus on reactive response after attacks have happened. This gives adversaries a significantly asymmetric advantage such that they have sufficient time to probe and learn our systems, and then prepare and launch attacks decisively to achieve their objectives within a short period of time, leaving little opportunity for defenders to defeat the attack actions.

To reverse defenders' disadvantages, the key change required is to engage with adversaries in the early stage of their cyber kill chain in order to disrupt and disable potential attacks. Recent research on the proactive defense concept has led to two major mechanisms: (i) moving target defense (MTD[2,3]), which increases complexity, diversity, and randomness of the cyber systems to disrupt adversaries' reconnaissance and attack planning, and (ii) cyber deception,[4–6] which provides plausible-looking yet misleading information to deceive attackers. Both schemes offer complementary approaches to defeat attack actions.

Although deception has been used since early ages of human history, its adoption into cyberspace has been mostly recent. In the late 1980s, Stoll[4] first discussed how to use deceptive techniques to trace intruders for computer security. The concept of a deception-based honeypot followed. To attract potential attackers, honeypots masquerade themselves as service hosts that could be potentially exploited. By collecting and recording information detailing the methods used in attackers' attempts to compromise honeypots, defenders can use the learned knowledge to enhance system security. More recently, a number of deception-based security designs[6,7] were introduced to confuse or mislead attackers. These cyber deception approaches have shown great promise in disrupting the cyber kill chain at its early stage.

In this article, we introduce cyber deception as an emerging proactive cyber defense technology. Because the research on cyber deception is still nascent, our objective is to provide a high-level overview of its life cycle, analyze unique aspects of cyber deception in comparison to nondeceptive approaches, and outline research challenges to stimulate more research going forward.

Formal View of Deception

Life Cycle of Cyber Deception

One of the most widely adopted definitions of cyber deception, proposed by Yuill,[5] is "planned actions taken

 Copublished by the IEEE Computer and Reliability Societies

to mislead and/or confuse attackers and to thereby cause them to take (or not take) specific actions that aid computer-security defenses." The essential parts in cyber deception include crafted information by the defender (that will be used to mislead) and wrong actions taken by the adversary as a result of the deception.

Figure 1 shows the life cycle of cyber deception at the conceptual level. Like traditional deception in the physical world, cyber deception is a revolving two-step action.

- In step 1, the defender will collect as much intelligence on the adversary as possible, in order to derive an estimate of the adversary's intent, capability, and decision process. This is fundamental to the success of cyber deception because without a solid situational understanding of potential attackers, an ad hoc deception scheme is no better than a random shot into the dark.
- In step 2, based on the understanding of the adversary, an actual deception scheme will be carefully crafted to manipulate and mislead the adversary. Key compositions of a cyber deception scheme can contain a combination of true and fabricated information and involve various deception techniques.

Through multiple rounds of engagements, the defenders' knowledge of the adversarial state (intent, capability, and decision process) may improve over time. In addition, deceptive actions by the defenders can be progressive as well. The defenders may choose to project additional deception information to adjust or reinforce their earlier deception schemes. In practice, a successful cyber deception will not likely be a one-time play, but instead a continuous multiround process that ties in both defenders and attackers.

Deception Formulation: Understand the Adversaries

Perceiving information learned from the adversary leads to the establishment of a good mental state model[8] that can help defenders identify the intent of adversaries, estimate their capabilities, and reason about potential attack actions. A good mental state model presents a good picture of the adversary that can potentially answer why, what, and how questions associated with observed attacker actions, which is a key requirement of any successful deception design. Early work by Daniel Dennett[9] on the Intentional Systems Theory attempted to explain an entity's behavior based on its beliefs and desires. He identified three levels of abstraction when we understand, explain, and predict behaviors:

- *Physical stance*: It is based on the explicit knowledge of the physical constitution and the physical laws that govern systems or the world. For example, if we know that an attacker may learn the physical location, hardware type, and power consumption profile of a server, we can infer that the attacker may be able to estimate the processing capability of the server and who it may belong to.
- *Design stance*: Without the need of knowing physical laws, design stance is based on the knowledge of the system's design. A system can be predicted to work as it is initially designed for. For example, if we know that an attacker can observe the traffic flows in and out of the target server, then the attacker may be able to infer the type of service being offered, clients that may be using the system, and the server's potential peers.
- *Intentional stance*: The behavior or action of an entity is governed by its intentional states (mental states), which are driven by beliefs, desires, intent, and motivation. In the above example, based on the knowledge of what the adversary may already know, if we believe that the adversary's intent is to steal information, we can potentially plant misleading or fake information on the server or in the server's traffic flows that the adversary has been targeting.

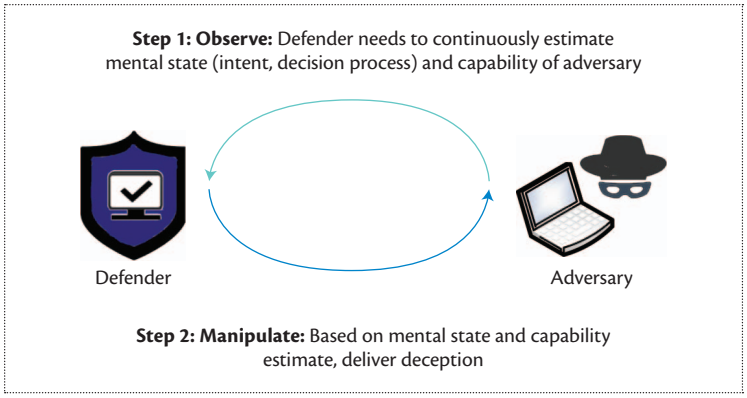


Figure 1. Life cycle of cyber deception.

If we adopt the Intentional Systems Theory to model an attacker's mental state, we assume that the attacker's plan of action follows his/her physical, design, and intentional stances. The task to understand the adversarial mental model relies on identifying the intentional stances, along with the design and physical stances of adversaries. Based on such understandings, defenders can attempt to manipulate adversaries' mindsets through the introduction of biases. A *cognitive bias*[10] represents the deviation from common sense (normal) or rational judgment and decision processes. Cognitive biases usually arise from information bias, where misinformation can be intentionally crafted and used to cause cognitive biases. Thus, the essence of deception is to understand

Table 1. Defender actions based on information availability.

		Adversary	
		Known	**Unknown**
Defender	**Known**	Undeniable truth, fact; selectively released truth. (Defender action: leverage, selectively release.)	Target of reconnaissance and information collection. (Defender action: deceit, protect, denial, selectively release.)
	Unknown	Dangerous area. Adversary has the advantage. Defender should strategically minimize this area.	Dark space.

an adversary and then to introduce cognitive biases in order to mislead. For the aforementioned server attack example, decoy servers with a typical server hardware profile (physical stance) and fabricated traffic patterns (design stance) could be introduced to divert attention. In addition, different types of service traffic flows could be fabricated at both real and decoy servers in order to introduce cognitive biases to the attackers such that the real server could be potentially camouflaged.

Deception Schemes and Common Actions

Although the key task for the cyber defender is to hide information from being disclosed, deception-based defense schemes depend on "misinformation" disclosure in order to deceive. To create believable "misinformation," the defender first needs to establish a situational awareness of what exactly the defender knows and does not know, as well as of how much the adversaries know. Then, based on this knowledge, the defender will plan strategically to selectively combine both true and fictitious information that ultimately can lead to adversaries' cognitive biases. A well-crafted deception scheme may contain disclosure of selected truth to convince the adversaries so that the overall cyber deception goal can be achieved.

The success of the deception scheme relies on defenders' asymmetric advantage over attackers: defenders know more! Observing, analyzing, and understanding the mental state (step 1 of any deception game) is the key to successful deception action in step 2. Based on the status of the information known or unknown to both the defender and the attacker, Table 1 lists the possible actions that the defender may take. While traditional defense focuses on information hiding, deception-based defense may use a combination of true information hiding and fake information disclosure to protect critical information and mislead the adversary at the same time. For information known to both the defender and the attacker, its selected partial disclosure is often combined with fake information release to make a deception scheme more convincing.

Any successful deception relies on the asymmetric information advantage over adversaries. Defenders should always try to minimize or eliminate their "blind spots" where they may not know the information that adversaries know, as listed in row 2 of Table 1. When the defender's advantage is reversed, the foundation for any deception scheme is destroyed.

Unique Aspects of Cyber Deception

In this section, we present the unique aspects of cyber deception, including key differences of cyber deception from non-cyber deception and a comparison to MTD, another major proactive defense strategy.

Key Differences from Non-Cyber Deception

Non-cyberspace deception includes both physical domain deception and social domain deception (that is, inter-human deception, including fraud). Cyber deception may share common traits of non-cyber deception and could be linked to non-cyberspace deception. For example, some attacks, such as advanced persistent threats, can contain physical, social, and cyber elements that maximize attack effectiveness. Cyber deception also has several unique aspects in terms of time, space, and speed constraints, as shown in Table 2.

From the time constraint perspective, things happen in chronological order in the physical world. Both physical and interpersonal deception schemes have to follow chronological order because it is not possible to change the physical world time arbitrarily. In cyberspace, an individual device or a network quite often relies on its own clock for time reference. Unless a strict time synchronization with a trusted global source is enforced, it is possible (although not always easy) to manipulate local clock settings such that a deception scheme can exploit "going back into the past." The chronological order of activities could potentially be altered among entities whose clocks were manipulated. This is the first unique aspect of cyber deception: when strict time synchronization is not enforced, time manipulation is quite possible.

In physical space, deception schemes normally have to follow physics principles in order to be plausible. Laws of physics limit what physical world deception can achieve. A deception artifact has to mimic physical attributes of a real entity. For example, fake tanks used in

Normandy during the D-Day operation needed to have the same physical size and color of real tanks in order to fool Nazi air reconnaissance through visual and optical imaging analysis. These artifacts had to be carefully designed such that they would conform to all physical principles governing the operational environment. In addition, a deception design needs to consider the technical capabilities of its recipients. For example, it would be much harder to use simple fake tanks today because new types of sensors (such as infrared imaging) will easily identify a real tank from fake ones using advanced sensing capabilities. In contrast, cyber deception may not be constrained by the physical world principles. The requirement to follow physical principles comes into play only when hardware devices are part of the deception scheme, or it is anticipated that the potential deception target (the adversary) may observe the system through hardware side channels.

In the social domain, interperson deception needs to conform to the norms of human communication and social interactions, and follow common social and cultural practices. Social space deception often relies heavily on interhuman relationships, direct interactions, and in-person verbal and nonverbal communications. Quite often cultural aspects also need to be carefully considered in social space deception. In contrast, cyber deception is largely confined to cyberspace, and has a much weaker social interaction constraint compared to social space deception.

The speed of building and deploying deception schemes in both physical and social spaces is limited by physical world principles and human interaction rules. In contrast, the speed of changes in cyberspace can be extremely fast because normally there are few physical or social constraints unless substantial hardware setup is required. This is another unique characteristic of cyber deception. Generally for cyber deception, the most challenging and time-consuming part of its life cycle is the observation and estimation of the mental model of adversaries and the follow-on planning for deception. Once deception schemes have been crafted, deploying a scheme can be done fairly rapidly, and similarly switching between schemes.

Table 2. Differences among cyber, physical, and social deception.

	Cyber deception	Physical deception	Social deception
Time	Past/Current/Future (if we control an individual system's clock)	Current, and always happening in chronological order	Current, and always happening in chronological order
Space	Weaker constraints, just obey computing and networking practices	Bounded by physical-space laws and principles	Bounded by normal social interactions
Speed	Extremely fast (can be done at computing or network processing speed)	Bounded by physical-space laws and principles	Bounded by human interaction and communication limit
Sensing	Virtual/Indirect	Physical means	Social interactions

Comparison to Moving Target Defense

MTD is another major proactive cyber defense technique that is being actively researched by the community. MTD refers to the techniques that continuously change a system's attack surface through adaptation, thereby greatly increasing the uncertainty, complexity, and costs for the attacker. Compared to MTD, the fundamental idea of cyber deception does not focus on transforming our cyber systems continuously, but instead on distracting attackers' attentions away from critical assets by relying on carefully crafted misinformation in order to create cognitive biases and to mislead adversaries' actions so that potential attacks are rendered onto the wrong targets.[8] Table 3 summarizes the key differences between MTD and cyber deception approaches.

MTD and deception are complementary techniques that can be deployed by the defender at the same time, with different focuses but having a common goal to defeat attacks. In the following, we summarize potential advantages of cyber deception over MTD when practically deployed.

- *Reducing overhead and saving resources.* System adaptation schemes are often complex and require extensive computing, network, and hardware resources to accomplish. Compared to MTD, cyber deception may need no significant system update and can be made simple and effective when correctly set up.
- *Understanding adversaries.* Most security mechanisms are designed to create a boundary around cyber systems and aim to stop illicit access attempts. In addition, MTD attempts to change the system boundary dynamically without trying to understand where adversaries may attempt to penetrate. Through the engagement with adversaries, cyber deception allows us to gain a better knowledge on adversaries, which may not only help build deception schemes, but also make MTD techniques less ad hoc and much more effective against potential attackers.
- *Increasing the risk on the adversary's side.* Cyber deception relies on passive and proactive probing,

Table 3. Comparisons between MTD and cyber deception.

	MTD	Cyber deception
Technical approach	Increase system complexity and diversity so that the adversary's observation rate is slower than the system change rate. Main focus is to deny information.	Does not focus on dynamic system changes, but uses a combination of information disclosure and hiding that can help project fake information to mislead.
Human engagement	Typically not addressed	Active engagement to observe and estimate adversary state and to inject misleading information
Social/cultural influence	Typically not addressed	Has to be incorporated as part of "genuine" fake information projection
Information disclosure	Never, focus on denying information	Yes, but only applicable to nonessential, selected truth combined with fake information projection

observing, and learning of adversaries. The possibility of deploying deception-based approaches may deter attackers who are not willing to take the risk of being exposed, analyzed, and further deceived. In contrast, MTD defense is weaker on deterrence from this perspective. Attackers may try all possible methods without worrying about being observed and misled.

The key message is that while MTD focuses on changing our system for defense, cyber deception aims at changing the adversary (its attack plan) to achieve the same objective. Typically, MTD relies on superior technology to win. Cyber deception, on the other hand, relies on more clever human engineering.

In this article, we provided an overview of emerging cyber deception research and identified the life cycle model and key characteristics of cyber deception. Although the use of deception to enhance cyber defense has shown a number of interesting and promising applications, substantial new research is needed to drive the area forward.

- *More accurate adversarial model.* Successful cyber deception schemes depend on a good understanding of adversaries, which is a challenging task. A well-defined adversarial model that incorporates mental state estimation is critical for both formulating a deception scheme and evaluating its effectiveness.
- *Continuous, multiround engagement.* Learning adversarial intent, capability, and methods requires continuous direct and indirect engagement. It is usually hard and sometimes impossible to collect information and learn about a potential attacker with whom we are not engaged. Honeypot or honeynet-like systems may attract adversaries and allow us to engage. The interactions enabled by honey systems may offer a valuable opportunity to learn the intent, capability, and potential goal of adversaries, and help us create an initial deception scheme and make follow-on adjustments.
- *Manipulation of adversarial mind.* The ultimate goal of cyber deception is to introduce cognitive biases to adversaries in order to manipulate their decision process and to mislead them into wrong decisions. We need to leverage advances in other disciplines (such as cognitive science, decision sciences, and control theory of systems) that can help us improve effectiveness in manipulating adversaries.
- *Usability analysis and quantification.* Regular users are a part of information systems. A successful deception scheme needs to ensure that it has the minimum impact on normal users. Deception metrics need to incorporate usability as part of its performance evaluation benchmark.
- *Combining deception and MTD approaches.* While MTD adapts a cyber system to increase its diversity and complexity in order to make it harder and more costly for adversaries to attack, cyber deception can be combined with MTD as a complementary method by using misinformation to lead adversaries into wrong actions and to drain their resources. More research is essential to understand how to combine both proactive techniques seamlessly and how to maximize proactive protection.

We envision that, going forward, a well-defined framework of cyber deception with detailed, specific application domains requires collaborative research involving computer and network security, control theory, as well as human cognitive science and psychology. Advances made in cyber deception research will lead to highly effective techniques for proactive cyber defense. ■

References

1. C. Croom, "The Cyber Kill-Chain: A Foundation for a New Cyber-Security Strategy," *High Frontier*, vol. 6, no. 4, 2010, pp. 52–56.
2. M. Crouse, B. Prosser, and E.W. Fulp, "Probabilistic Performance Analysis of Moving Target and Deception Reconnaissance Defenses," *Proc. of ACM MTD*, 2015.

3. S. Jajodia et al., *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54, Springer Science & Business Media, 2011.
4. C.P. Stoll, "The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage," Doubleday, 1989.
5. J.J. Yuill, *Defensive Computer-Security Deception Operations: Processes, Principles and Techniques*, ProQuest, 2006.
6. A. Juels and R.L. Rivest, "Honeywords: Making Password-Cracking Detectable," *Proc. of ACM CCS*, 2013.
7. N.C. Paxton et al., "Utilizing Network Science and Honeynets for Software Induced Cyber Incident Analysis," *Proc. of HICSS*, 2015.
8. M.H. Almeshekah and E.H. Spafford, "Planning and Integrating Deception into Computer Security Defenses," *Proc. of NSPW*, 2014.
9. D. Dennett, "Intentional Systems Theory," *The Oxford Handbook of Philosophy of Mind*, 2009, pp. 339–350.
10. M.G. Haselton, D. Nettle, and D.R. Murray, "The Evolution of Cognitive Bias," *The Handbook of Evolutionary Psychology*, 2005.

Cliff Wang graduated from North Carolina State University with a PhD in computer engineering in 1996. He has been carrying out research in the area of computer vision, medical imaging, high speed networks, and most recently information security. Since 2003, Wang has been managing extramural research portfolio on information assurance at the US Army Research Office. In 2007, he was selected as the director of the computing sciences division at ARO while at the same time managing his program in cybersecurity. Wang also holds adjunct full professor appointment at the Department of Electrical and Computer Engineering at North Carolina State University. He was elected to IEEE Fellow in 2016. Contact him at cliffwang@ncsu.edu.

Zhuo Lu is an assistant professor at the Department of Electrical Engineering, University of South Florida. He is also affiliated with the Florida Center for Cybersecurity. Lu received his PhD from North Carolina State University in 2013. His research has been mainly focused on modeling and analytical perspectives on communication, network, and security. His recent research is equally focused on practical and system perspectives on networking and security. He is an IEEE member. Contact him at zhuolu@usf.edu.

Editors: Heather Richter Lipford, richter@uncc.edu | Jessica Staddon, jessica.staddon@gmail.com

The Privacy Paradox of Adolescent Online Safety:

A Matter of Risk Prevention or Risk Resilience?

Pamela Wisniewski | University of Central Florida

Networked technology is an ever-present force in the lives of nearly all teens; according to Pew Research, 92 percent of teens (ages 13 to 17) in the US go online daily, 73 percent have access to smartphones, and 71 percent engage in more than one social media platform.[1] Some argue that the prevalent use of mobile smartphones and social media has created a "tethered" society, which facilitates a wide array of new social interactions but also amplifies online risks. The Crimes against Children Research Center reports that 23 percent of youth have experienced unwanted exposure to Internet pornography, 11 percent have been victims of online harassment, and 9 percent report receiving unwanted sexual solicitations online.[2]

While concerning, these exposure rates do not indicate that online risks are an epidemic, nor do they necessitate moral panic; in fact, there is little evidence that online risk presents more harm than the risks teens typically encounter offline. Yet, the fear that teens will certainly fall victim to unthinkable online dangers persists, shaping the technologies designed to keep teens safe online. This article challenges the current solutions for protecting teens online and suggests a paradigm shift toward empowering teens to be agents of their own online safety by teaching them how to self-regulate their online experiences.

A Paradox between Safety and Control

The "privacy paradox" traditionally points to the discrepancy between individuals' stated privacy concerns versus their conflicting disclosure behaviors, such as oversharing on social media. However, when Barnes first coined the term over a decade ago,[3] she specifically referred to the online dangers (that is, sexual predation) posed to teens due to their over-willingness (that is, lack of concern) to share personal information via social media combined with their paradoxical desire to protect their publicly posted private thoughts from their parents.

The assumption that teens are at risk online because of their poor disclosure decisions is prevalent in the literature, resulting in a body of research focused on increasing teens' concern for privacy as a way to reduce their online disclosures, thereby protecting them from encountering online risks. Ironically, this "privacy as prevention" approach to online safety has resulted in privacy-invasive tools that now allow parents to monitor and restrict their teens' online behaviors,[4] which only exacerbates the privacy tensions between parents and teens.[5]

On one hand, we are telling teens that they need to care about

 Copublished by the IEEE Computer and Reliability Societies

their online privacy to stay safe, and on the other, we are taking their privacy away for the sake of their online safety. This catch-22 poses a "new" privacy paradox for our youth that needs to be addressed when designing future technologies to protect teens from online risks. In her book, *It's Complicated: The Social Lives of Networked Teens*, boyd argues, "as a society, we often spend so much time worrying about young people that we fail to account for how our paternalism and protectionism hinders teens' ability to become informed, thoughtful, and engaged adults."[6]

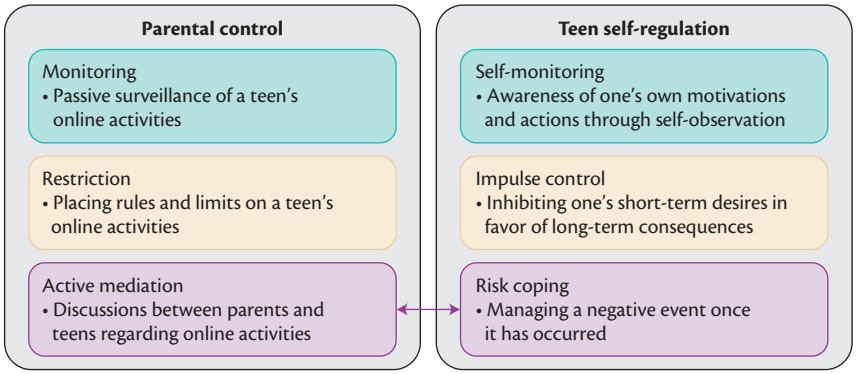


Figure 1. Conceptual framework of Teen Online Safety Strategies (TOSS).

My past work illustrates why paternalistic and abstinence-oriented approaches to adolescent online safety simply fall short. In traditional families, communication regarding the risks teens experience online is particularly poor; parents tend to be overly judgmental and overreact when teens disclose their online risk experiences, making the problem worse.[5] Overall, restrictive parenting practices have a suppressive effect, reducing risks but also opportunities for moral growth and beneficial online engagement. Parent-focused approaches to adolescent online safety also assume a significant level of privilege; teens, especially those who are most vulnerable to online risks (for instance, foster youth), often do not have parents who are actively engaged in ensuring their online safety. Finally, such approaches do not teach teens how to effectively protect themselves online.[7]

Teens, Privacy, and Online Safety

The assumption that teens lack the ability to make calculated privacy decisions online has been debunked; teens do take protective measures against online risks and value their privacy, but they also value the social benefits of engaging online.[6,7] As such, teens exhibit a markedly different privacy calculus than adults; they treat "risk as a learning process," taking protective measures to recover once disclosures have escalated to the point of potentially harmful interactions.[7] Some level of risk taking and autonomy seeking is a natural and necessary part of adolescence, and preventing such experiences may actually stunt developmental growth as teens strive to individuate themselves from their parents. Thus, new interventions for adolescent online safety need to reflect how teens manage their online privacy, not how we do as adults. Therefore, the long-term goal of design-based interventions for adolescent online safety should be to teach youth how to effectively manage online risks as they transition into adulthood, not just to shield them from online risks.

Moving toward Risk-Resilient Teens

Adolescent resilience theory is a strength-based approach developed to explain divergent outcomes related to various teen risk behaviors, including substance abuse, violent behavior, and sexual promiscuity.[8] Resilience is an individual's ability to thrive in spite of significant adversity or negative risk experiences. Our research has confirmed that resilience plays a significant role in protecting teens from the negative effects of Internet addiction and online risk exposure.[9] Teens are often able to cope and resolve negative online experiences without intervention from their parents, even benefiting from experiencing some level of online risk by learning from their mistakes and developing crucial interpersonal skills, including boundary setting, conflict resolution, and empathy.[7] Therefore, we developed a new conceptual framework of Teen Online Safety Strategies (TOSS), which attempts to shift the balance and rectify the privacy paradox between parents and teens. It was theoretically derived to illustrate the tensions between parental control and teen self-regulation when it comes to teens' online behaviors, their desire for privacy, and their online safety (Figure 1).

In the TOSS framework, parental control strategies for online safety include:

- *Monitoring*: passive surveillance of a teen's online activities,
- *Restriction*: placing rules and limits on a teen's online activities, and
- *Active mediation*: discussions between parents and teens regarding their online activities.

Teen self-regulation strategies were drawn from the adolescent

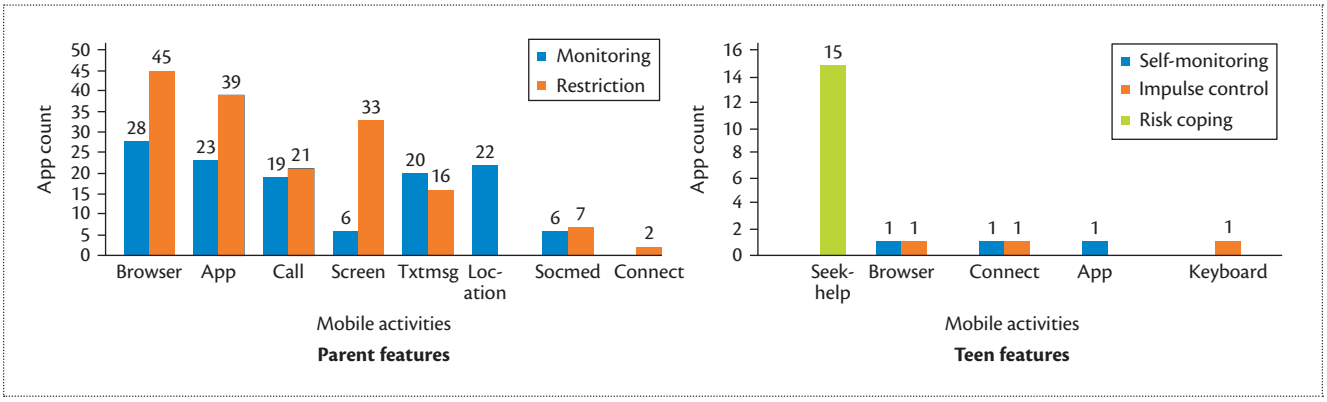


Figure 2. Parent versus teen features within mobile online safety apps.

developmental psychology literature, which considers self-regulation a "resiliency factor" that protects teens from offline risks by modulating emotions and behaviors through monitoring, inhibition, and self-evaluation. Teen self-regulation strategies include *self-monitoring*, *impulse control*, and *risk coping*.

For teens to effectively self-regulate their online behaviors, they must be aware of their own actions through self-observation (that is, self-monitoring). Impulse control aids in self-regulation by inhibiting one's short-term desires in favor of positive long-term consequences. Risk coping is a component of self-regulation that occurs after one encounters a stressful situation, which involves addressing a problem in a way that mitigates harm. Because risk coping is influenced by teens' own appraisals of online risk as well as that of their parents', the TOSS framework makes an explicit association between active parental mediation and teen risk coping.

A Critical Assessment of Where We Are Now

In two recent studies, we applied the TOSS framework to better understand the commercially available technical offerings that support adolescent online safety, and what teens thought about these applications. First, we analyzed the features within 75 commercially available mobile apps on Android Play that had the primary or secondary purpose of promoting teen mobile online safety.[4] By downloading and exploring the apps, we identified 42 unique features (for instance, monitoring and restricting web browsing, app installations, calls, screen time, and so on) with 382 instances of these features being supported across the apps in our dataset.

An overwhelming majority of features (89 percent) within these apps supported parental control through monitoring (44 percent) and restriction (43 percent), as opposed to facilitating parents' active mediation or supporting any form of teen self-regulation. Many of the apps were extremely privacy invasive, providing parents granular access to monitor and restrict teens' intimate online interactions with others, including browsing history, the apps installed on their phones, and the text messages teens sent and received. Teen risk coping was minimally supported by an "SOS feature" that teens could use to get help from an adult (Figure 2).

Based on these results, we argue that existing apps do not reflect positive family values (for instance, trust, respect, and empowerment) that meet the needs of teens or parents. Essentially, we are telling teens that they cannot be trusted and that we, as adults, must protect them from any dangers they may encounter in online spaces. Such fear-based messages do little to empower teens and, arguably, to keep them safe online.

In a follow-up study, we analyzed 736 reviews of these parental control apps that were publicly posted by teens and younger children on Google Play.[10] We found that the majority (79 percent) of children overwhelmingly disliked the apps, while a small minority (21 percent) of reviews saw benefits to the apps. Children rated the apps significantly lower than parents; the mean difference between parents' mean score was 1.87 (95 percent CI [confidence interval], 1.76 to 1.98) higher than children's mean score, $t(793.34) = 33.77, p < 0.05$).

We conducted a thematic content analysis, which uncovered that teens, and even younger children, strongly disliked these apps because they felt that they were overly restrictive and invasive of their personal privacy, and negatively impacted their relationships with their parents. Many reviews suggested that the apps were so restrictive that the children could no

longer accomplish everyday tasks, such as doing their homework:

> *"It doesn't just protect you from the porn and stuff, it protects you from the whole internet!! It wouldn't let me look up puppies!...If I can give it less than a star I would!!" —One Star, Net Nanny for Android, 2014*

They also thought that the online safety apps completely violated their personal privacy and equated the apps to a form of parental stalking:

> *"Fantastic. Now now my mom is stalking me. I have nothing to hide. You can always just ask to go through my phone. Too invasive and down right disrespectful. Thanks for the trust, mom." —One Star, MamaBear Family Safety, 2014*

These children were very vocal in their opinions about the apps not aligning with good parenting techniques, such as communicating with them or trusting them to make good decisions:

> *"Seriously, if you love your kids at all, why don't you try communicating with them instead of buying spyware. What's wrong with you all? And you say we're the generation with communication problems." —One Star, SecureTeen Parental Control, 2016*

Comparing our results through the TOSS framework (Figure 3), the reasons why teens and younger children disliked these apps aligned directly with the TOSS dimensions for parental control—which were the online safety strategies that our earlier study found were well-supported by these apps. We found that reviews were more positive when children felt that the apps afforded them more agency (that is, self-regulation) or improved their relationship with their parents (that is, active mediation). For instance, some children found apps useful when they helped them control unhealthy or addictive behaviors (that is, impulse control) or gave them more awareness of their unhealthy behaviors (that is, self-monitoring). They were open to using online safety apps when they saw direct benefits, such as managing unhealthy behaviors.

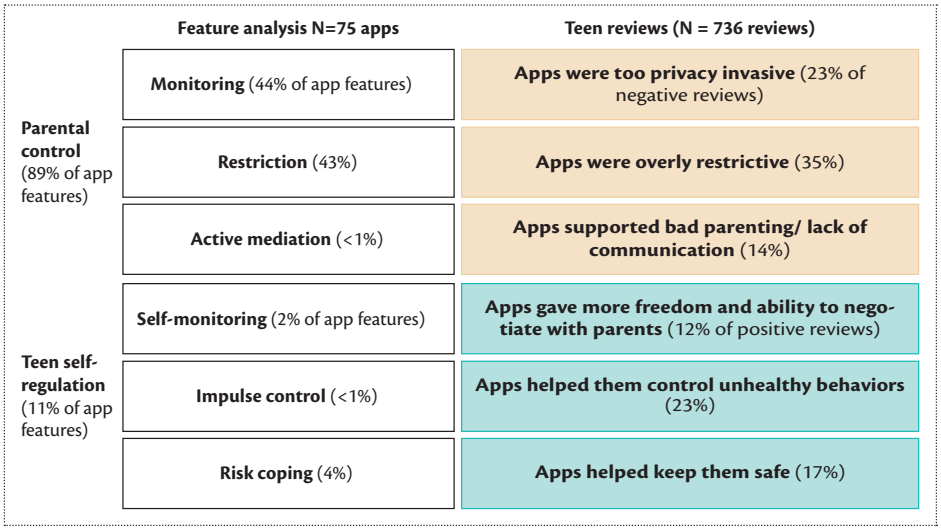


Figure 3. Summary of research findings.

A takeaway from this research is that, as researchers and designers, we should consider listening to what teens have to say about the technologies designed to keep them safe online and conceptualize new solutions that engage parents and respect the challenges teens face growing up in a networked world.

A Possible Path Forward

During adolescence, teens need personal and psychological space for positive development; privacy also becomes very important in terms of the parent–teen relationship to build trust and allow teens a level of personal autonomy. To compromise on solutions that may meet both parents' desire to keep their children safe and teens' desire to uphold personal privacy, we make a number of design recommendations targeted toward app designers to increase teen adoption and acceptance of mobile safety apps by thinking of teens as their end users.

Empower Teens as End Users

Encourage teens to use mobile apps to self-regulate their own behaviors (as opposed to being forced to use an app by their parents). Few teens are going to opt to install an app that explicitly says that it is for "parental control," which was the most common moniker in commercially available apps. Prompting teens to use mobile online safety apps themselves and engaging with teens directly as end users may empower teens by giving them more agency and choice, thereby increasing their sense of personal autonomy and control.

Use a Teen-Centric Approach to Design

Provide features teens find personally beneficial. We should leverage user-centered techniques to better understand what mobile safety features teens would actually find useful. Instead of assuming that teens are inherently risk seeking, a

more nuanced approach would be to ask them in what ways they feel that they need to be kept safe. For instance, we found that some teens liked apps that helped them disconnect from their phones or reduce other problematic behaviors. Therefore, teens may prefer "personal assistant"–type features that assist (not restrict) them in being more aware of their unhealthy behaviors and changing them without parental intervention. These features could keep track of teens' activities via their smart devices and "nudge" them whenever an inappropriate behavior is detected.

Design for Safety with Privacy in Mind

Create online safety apps that employ a level of abstraction to give parents helpful meta-level information regarding teens' mobile activities instead of full disclosure of what teens do from their mobile phones. For example, an app may provide parents a high-level summary of who their teen is engaging with via their mobile device and how often, as opposed to divulging the content of every conversation.

Help Teens Communicate with Their Parents

Provide features so that teens can negotiate with parents. In cases when teens' perceptions of appropriate online behaviors conflict with their parents', it would be helpful for online safety apps to provide flexible parental controls that support and are more contingent on appropriate contexts of use, giving teens the ability to negotiate with their parents in particular circumstances. For instance, more app designs may consider implementing features similar to the "reward time system" offered by the Screen Time Companion app that allows teens to get extra time if they meet certain criteria specified by their parents. Reward systems are more contextualized restraints because they provide positive reinforcement and allow teens to earn privileges, as well as their parents' trust.

By taking a more "teen-centric" instead of a "parent-centric" approach to adolescent online safety, researchers and designers can help teens foster a stronger sense of personal agency for self-regulating their own online behaviors and managing online risks. Technology should support teens in their developmental goals, including information seeking, learning about rules and boundaries, and maintaining social relationships, in addition to keeping them safe from online risks. However, this goal will only be accomplished once we listen more intently to teens as end users. As such, we call for new design practices that are more teen-centric and place value on online safety as an integral part of their adolescent and developmental growth, teaching teens the skills and giving them to confidence to engage safely and smartly with others through the Internet. ■

References

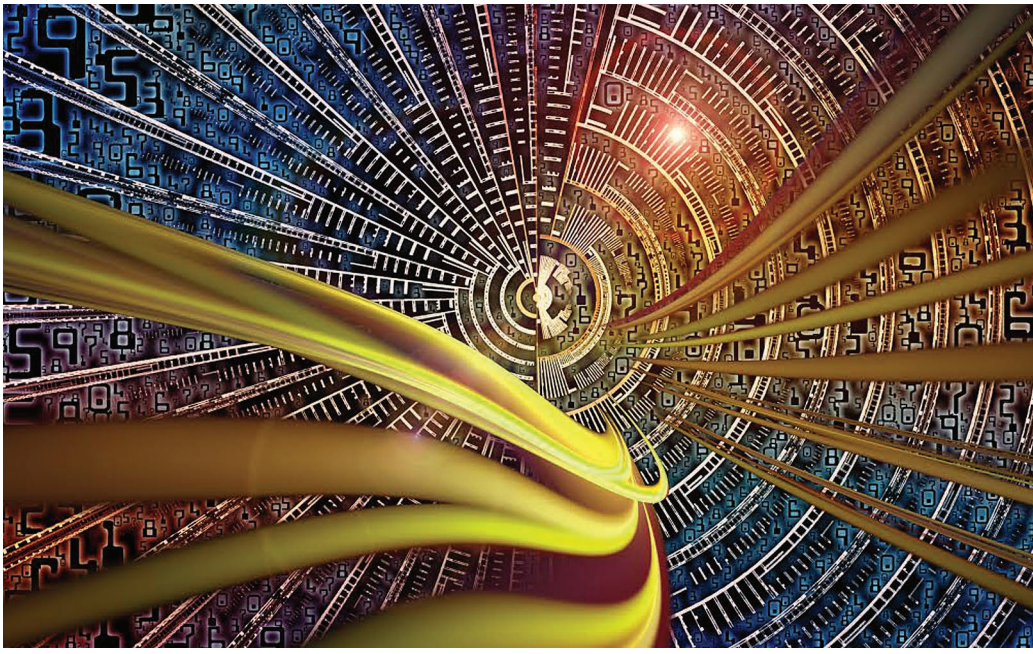
1. A. Lenhart, *Teens, Social Media & Technology Overview 2015*, Pew Research Center: Internet, Science & Tech, 2015.
2. L.M. Jones et al., "Trends in Youth Internet Victimization: Findings from Three Youth Internet Safety Surveys 2000–2010," *Journal of Adolescent Health*, vol. 50, no. 2, 2012, pp. 179–186; https://doi.org/10.1016/j.jadohealth.2011.09.015.
3. S.B. Barnes, "A Privacy Paradox: Social Networking in the United States," *First Monday*, vol. 11, no. 9, 2006; https://doi.org/10.5210/fm.v11i9.1394.
4. P. Wisniewski et al., "Parental Control vs. Teen Self-Regulation: Is There a Middle Ground for Mobile Online Safety?," *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, 2017, pp. 51–69.
5. P. Wisniewski et al., "Parents Just Don't Understand: Why Teens Don't Talk to Parents about Their Online Risk Experiences," *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, 2017, pp. 523–540.
6. d. boyd, *It's Complicated: The Social Lives of Networked Teens*, Yale University Press, 2014.
7. P. Wisniewski et al., "Dear Diary: Teens Reflect on Their Weekly Online Risk Experiences," *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 3919–3930.
8. F. Stevenson and M.A. Zimmerman, "Adolescent Resilience: A Framework for Understanding Healthy Development in the Face of Risk," *Annual Review of Public Health*, vol. 26, 2005, pp. 399–419.
9. P. Wisniewski et al., "Resilience Mitigates the Negative Effects of Adolescent Internet Addiction and Online Risk Exposure," *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015, pp. 4029–4038.
10. A.K. Ghosh et al., "Safety vs. Surveillance: What Children Have to Say about Mobile Apps for Parental Control," to be published in *Proceedings of the 2018 ACM Conference on Human Factors in Computing Systems*, 2018.

Pamela Wisniewski is an assistant professor at the University of Central Florida. Contact at pamwis@ucf.edu.

Editors: Melissa Dark, dark@purdue.edu | Jelena Mirkovic, mirkovic@isi.edu | Bill Newhouse, william.newhouse@nist.gov

Individualizing Cybersecurity Lab Exercises with Labtainers

Michael F. Thompson and Cynthia E. Irvine | Naval Postgraduate School



Hands-on laboratory exercises help students internalize knowledge so that it can be applied in new contexts. Hence, cybersecurity educators try to create lab exercises that allow students to explore systems, yet provide sufficient guidance so that students achieve the desired learning objectives without becoming lost in minutiae. Challenges associated with developing such exercises include creating and supporting each lab, ensuring students do their own work, and grading exploratory activities.

In many cases, providing labs is difficult because access to physical lab computers—or remote access to institutionally or other centrally provisioned and managed resources—is not practical. Binding students to centralized servers can make self-paced, intermittent activity more difficult, yet, lacking institutional IT equipment and staff, instructors may not be able to present easily managed and deployed fine-tuned lab environments. However, if students run lab exercises directly on their own computers, other problems arise: the results produced may vary from student to student depending on software installed on the computer used, and all the tools required for an exercise may not even execute on certain platforms. The solution is tailored cybersecurity lab environments that eliminate divergent results caused by software differences. This can be achieved by providing students with virtual machine (VM) images containing lab-related software.[1] Students then run VMs on their personal computers or on institutionally provided computers. Through use of VMs, variations in the results among students can be largely limited to hardware performance differences.

But, use of VMs on student computers has several drawbacks. First, exercises involving two or more networked computers require multiple VMs, the hosting of which is beyond the performance capabilities of many student computers. Also, different labs may rely on mutually incompatible configurations or software packages, thus requiring students to either perform complex provisioning steps or to install separate VMs for each lab. The provisioning and administration of the execution environments required by different labs can become a significant distraction and source of frustration for both students and instructors.

Another challenge is that, regardless of how they are provisioned, cybersecurity lab exercises are often susceptible to students' sharing solutions and cribbing from each other's lab reports. Use of VM images on individual student computers complicates schemes designed to verify student performance of their own lab exercises, for example, logging

1540-7993/18/$33.00 © 2018 IEEE Copublished by the IEEE Computer and Reliability Societies March/April 2018

and audit features that might be part of a remotely accessed cyber range. Student actions on VMs can be logged; however, use of a single VM for multiple labs would require some method to distinguish the artifacts of different labs. The alternative of allocating each lab to a distinct VM image can be prohibitive in terms of network bandwidth and disk storage on the student computer.

The last challenge is encouraging students to explore the lab environment while providing instructors with a simple way to determine that students have achieved expected milestones. How can students "show their work"? How can instructors observe what students have done and provide advice if they are stuck, yet not have to stand over the students while they complete the entire exercise?

Labtainers: A Practical Solution Using Docker Containers

Labtainers is a framework for developing and deploying Linux-based labs involving multicomponent network topologies all hosted entirely on modestly provisioned student computers. Our initial emphasis is on cybersecurity. Docker containers[2] are used to standardize complete lab execution environments, thereby reducing lab setup and configuration distractions. By using containers, labs can incorporate complex topologies without suffering the overhead of running multiple VMs. The Labtainer framework supports automated assessment of student work and allows lab exercises to be individualized for each student, thus discouraging the appropriation of others' work.

The use of Docker containers simplifies the Labtainer approach to individualizing student labs and recording student activity for later assessment by instructors. The framework automatically collects artifacts from a student lab environment into an archive file that the student forwards to her instructor. Here we describe strategies for ensuring that the artifacts in the archive file are the result of that student's efforts. We present these strategies in the context of two example Labtainer exercises. The first provides an introduction to network traffic analysis using *tshark*, and the second employs the *nmap* utility to locate a selected network service.

The Labtainer framework supports three types of users. *Lab designers* are responsible for creating laboratory exercises so that they meet intended learning objectives. Each lab designer determines if and how the lab is parameterized and whether automated assessment will be supported. *Instructors* assign labs to students and assess their work. Instructors may or may not work with lab designers to create exercises. *Students* perform the laboratory exercises. They are oblivious to the underlying framework that configures and individualizes their labs and that gathers artifacts required for assessment.

Target Lab Context and Automated Assessment

Students start Labtainer exercises by executing a Python script on a Linux host, typically a VM. The script augments the Linux host environment with one or more Docker containers and a set of virtual terminals. Students use the virtual terminals to interact with the containers, which from the students' vantage point appear to be independent computers. The execution environment within each container is prescribed by the designer of the lab. In the degenerate case where the lab designer provides only a name for the lab, the environment seen by the student will be a *bash* shell on what appears to the student to be an Ubuntu Linux system. The Labtainer framework allows the lab designer to select from a variety of Linux distributions for each container and to include software packages and configuration settings as appropriate for the lab. Designers define virtual networks and the connections among containers. The student sees the resulting network topology and has virtual terminals connected to only those containers indicated by the designer.

After the student performs the lab exercise, she runs another Python script that terminates the lab on the Linux host. This results in the collection of artifacts from her lab activity. She then provides the resulting archive to the instructor. The instructor can review these artifacts on similarly provisioned Docker containers. The framework includes tools for the lab designer to specify expected attributes of the student artifacts, which are then automatically assessed and summarized for the instructor. Labtainer support for consistent execution environment provisioning and automated assessment of student work is described in detail elsewhere.[3,4]

Attribution through Lab Individualization

Several approaches to ensuring the individuality of student work are possible in the Labtainer framework: *watermarks*, *per-student artifacts*, and *per-student solutions*. Within a particular lab exercise, these can be used separately or in combination.

Per-Student Watermarks

When a student starts a lab, the Labtainer framework incorporates a student-supplied email address into a seed for generation of pseudorandom values. A watermark file is automatically created for each student lab, and this file becomes one of the artifacts in the student's archive. The watermark value is validated as part of the assessment process initiated by the instructor.

This simple strategy ensures (albeit weakly) that the archive provided by the student originated with the student who started the lab. As will be seen in subsequent sections, the Labtainer framework allows lab designers to improve on the assurances provided by the watermarks.

The simple watermark check inherent in all Labtainer exercises is readily bypassed by replacing a single file in the archive, perhaps by an automated script shared among students, effective on all Labtainer exercises. Variations on the watermark strategy can be defeated by correspondingly advanced automations, for instance, scripts that replace individual artifacts such as the output of a program invocation. These automations become specific to the individual labs and thus require more effort by the benevolent cheater to build and maintain. A fundamental limitation on the robustness of the watermarking strategies is that the Labtainer framework does not keep secrets from the student environment. Our design explicitly avoided the large step in complexity inherent in maintaining secrets.

Additional assurances of the originality of student work rely on choices made by the lab designer. In the Labtainer framework, these schemes typically have one of two purposes. The first, per-student artifacts, provides further evidence that someone performed the exercise on a Labtainer instance that was initiated using the student's email address. This strategy causes selected artifacts generated by lab exercise steps to be individualized for each student. The second approach, per-student solutions, seeks to ensure that whoever performed the exercise did not simply reproduce dictated actions. In the Labtainer framework, these per-student solutions, when practical for a given lab, provide more assurance that students performed their own work.

Per-Student Artifacts

Introduction of per-student artifacts makes development of cheating automations more challenging, because individual artifacts themselves are tailored to the individual student. A simple example is an exercise that requires the student to display to standard output the content of a student-specific file on a server once a remote shell on the server is obtained. This would defeat an automation that simply inserts unmodified artifacts into the student's archive. While one can imagine correspondingly sophisticated automations that are informed by the particulars of the lab exercise, at some point, it becomes easier for enterprising students to publish and re-perform exact keystrokes necessary to create the desired artifacts. For some labs, the keystrokes problem can be addressed by per-student solutions.

A less trivial example of per-student artifacts is found in the Labtainier *pcap analysis* lab, in which students are introduced to basic network traffic analysis techniques using the tshark utility. Each student's Labtainer environment for this lab includes a *pcap* file tailored to that student. The pcap file is individualized by truncating a random quantity of "filler" packets from the start of a baseline pcap file. This results in packet numbers that are unique to the student, while the content of the non-filler traffic remains constant for all students. The student is required to display the single packet of a specific invalid login attempt. Hence, the output of the corresponding tshark command will include a student-specific packet number that can be deterministically reproduced by the assessment function in the instructor's environment.

Lab designers individualize labs using parameterization configuration files containing commands that cause the framework to replace symbols in selected files with random values derived from the student email address. For the pcap analysis lab, the target of replacement is a parameter passed to a utility that truncates the start of the pcap file. This *parameterization* utility is invoked the first time a student runs the lab, resulting in truncation of the pcap file seen by the student. (Note: students do not have to complete a lab in one sitting, and if the student wishes, the framework allows her to restart a lab from the beginning, with complete reinitialization and consistent personalization.) The configuration file entry shown in Figure 1 causes the symbol "START_FRAME" in the file "fixlocal.sh" to be replaced by a random value between 1 and 100.

```
FIRST_FRAME : RAND_REPLACE: .local/
bin/fixlocal.sh : START_FRAME : 1 :
100
```

Figure 1. Extract from parameterization configuration file.

Lab designers define automated assessment in assessment configuration files that identify student artifacts and their expected attributes. The pcap analysis lab assessment configuration files identify the standard input of the tshark command as an artifact of interest. The configuration file includes a directive to extract the "frame.number" filter argument provided to tshark. Labtainer assessment configuration file directives allow the designer to symbolically reference symbols named in parameterization configuration files. The expected value of the frame number is derived by subtracting the random value used during parameterization from the value of the frame number as it existed before the pcap file was truncated. The configuration file entry in Figure 2 subtracts from 190 the

```
view_frame = matchany : integer_
equal : (190-frame_num) : parameter.
FIRST_FRAME
```

Figure 2. Extract from assessment configuration file.

"frame_num" value found in a student's artifact and compares this to the random value that resulted from the directive in Figure 1.

Per-Student Solutions

The example pcap analysis lab is susceptible to one student providing another with the precise keystrokes needed to complete the lab—a problem associated with all labs that rely only on per-student results. Per-student solutions defeat rote repetition of keystrokes. An example is the *nmap-discovery* lab, which presents the student with a fictional scenario in which he is told that he has an account on an SSH server but is given only the server name (not the network address) and his password. The student uses nmap to locate the server IP address and to discover the SSH port number that the IT department had set to an arbitrary value. This exercise is individualized by assigning a random port number, within a range, to each student. Thus, rote keystroke repetition fails to complete the lab.

The parameterization configuration file for the nmap-discovery lab names symbols in Linux system services configuration files. These system files were modified by the lab designer to contain symbolic names in place of the SSH port numbers. These symbolic names are replaced during parameterization. As a result, system networking services on the configured container will listen to the individualized port number for SSH traffic.

In this particular lab, there is no need for assessment configuration directives to reference symbols in parameterization configuration files, because the student could not have SSH'd to the server unless the port number was discovered. This simplifies automated assessment and is in contrast to the previous example in which the assessment configuration files referenced the pcap truncation parameter.

The nmap-discovery lab automated assessment reveals whether the student was able to use SSH to connect to the target server and the number of times the student invoked the nmap command. The assessment configuration file identifies standard output from the SSH command as an artifact of interest—specifically any output that contains a constant string within a file present on the target SSH server. If this string appears in the artifacts, the student is assumed to have discovered the SSH port number.

Beyond the primary motivation of not rewarding rote replays of lab steps, per-student solutions have an advantage from the perspective of the lab designer. Because the parameterization need not be reproduced in the assessment step, the expected results as represented in the student artifacts can be confirmed without reference to any specific parameterized values generated for that student. Although making a tie between a parameterized value and the expected results in the assessment configuration files is often relatively straightforward, it can become tedious and error prone for some exercises. Consider a forensics-oriented lab that requires students to recover a deleted file from a virtual file system. A simple way to individualize the lab is to add a randomly determined number of filler files into the file system prior to creating the files of interest. The filler files force file offsets and inode numbers of the target files to be a function of the student's email address. Assessing per-student results for this lab (for instance, comparing student-provided inode values with expected values) is challenging because the effects of filler files on inodes and offsets depend on file system implementation vagaries, and these are not easily predicted. The assessment step for this forensics lab need not be concerned with what the values are for any given student; rather it can rely only on whether the deleted file was in fact recovered. Thus, a per-student solution assumes the student could not have recovered the file content without discovering the offset or inode specific to that student.

Status and Availability

More than 35 labs are available to students and instructors in the Labtainer framework (https://my.nps.edu/web/c3o/labtainers). Each includes a student lab manual, and most include automated assessment and perform per-student individualization. The website also includes a developer package for use by lab designers when creating new labs or transitioning existing labs into the Labtainer framework. The containers themselves are hosted on the public Docker hub (https://hub.docker.com) and are transparently loaded onto a student's computer when the corresponding lab is first started. The "Labtainer Lab Designer User Guide"[4] describes how lab designers can publish their own Labtainers-based labs on the Docker hub, thus making them available to their students.

Future work we hope to pursue on Labtainers includes GUI-based, integrated lab-authoring tools as well as additional features to support instructors. These include HTML-based reporting of student assessment results and integration into learning management systems. We would also like to convert our development and publishing work flow to a collaborative environment to simplify the integration of

contributions by a community of lab designers. ■

Acknowledgments

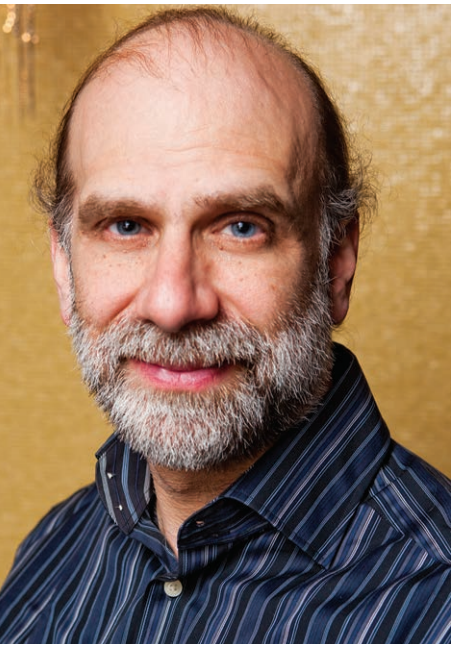
This work was supported by NSF grant DUE-1438893. The views expressed in this material are those of the authors and do not reflect the official policy or position of the National Science Foundation, Department of Defense or the US Government.

References

1. W. Du, "SEED: Hands-On Lab Exercises for Computer Security Education," *IEEE Security & Privacy*, vol. 9, no. 5, 2011, pp. 70–73.
2. A. Arvam, "Docker: Automated and Consistent Software Deployments," 27 Mar. 2013; https://www.infoq.com/news/2013/03/Docker.
3. C.E. Irvine et al., "Labtainers: A Docker-Based Framework for Cybersecurity Labs," *Proc. 2017 USENIX Workshop on Advances in Security Education*, August 2017.
4. M.F. Thompson, "Labtainer Lab Designer User Guide," 27 Oct. 2017; http://my.nps.edu/documents/107523844/109121513/labdesigner.pdf.

Michael F. Thompson is with Naval Postgraduate School. Contact him at mfthomps@nps.edu.

Cynthia E. Irvine is with Naval Postgraduate School. Contact her at irvine@nps.edu.



Bruce Schneier
Harvard University

Artificial Intelligence and the Attack/Defense Balance

Artificial intelligence technologies have the potential to upend the long-standing advantage that attack has over defense on the Internet. This has to do with the relative strengths and weaknesses of people and computers, how those all interplay in Internet security, and where AI technologies might change things.

You can divide Internet security tasks into two sets: what humans do well and what computers do well. Traditionally, computers excel at speed, scale, and scope. They can launch attacks in milliseconds and infect millions of computers. They can scan computer code to look for particular kinds of vulnerabilities, and data packets to identify particular kinds of attacks.

Humans, conversely, excel at thinking and reasoning. They can look at the data and distinguish a real attack from a false alarm, understand the attack as it's happening, and respond to it. They can find new sorts of vulnerabilities in systems. Humans are creative and adaptive, and can understand context.

Computers—so far, at least—are bad at what humans do well. They're not creative or adaptive. They don't understand context. They can behave irrationally because of those things.

Humans are slow, and get bored at repetitive tasks. They're terrible at big data analysis. They use cognitive shortcuts, and can only keep a few data points in their head at a time. They can also behave irrationally because of those things.

AI will allow computers to take over Internet security tasks from humans, and then do them faster and at scale. Here are possible AI capabilities:

- Discovering new vulnerabilities—and, more importantly, new types of vulnerabilities—in systems, both by the offense to exploit and by the defense to patch, and then automatically exploiting or patching them.
- Reacting and adapting to an adversary's actions, again both on the offense and defense sides. This includes reasoning about those actions and what they mean in the context of the attack and the environment.
- Abstracting lessons from individual incidents, generalizing them across systems and networks, and applying those lessons to increase attack and defense effectiveness elsewhere.
- Identifying strategic and tactical trends from large datasets and using those trends to adapt attack and defense tactics.

That's an incomplete list. I don't think anyone can predict what AI technologies will be capable of. But it's not unreasonable to look at what humans do today and imagine a future where AIs are doing the same things, only at computer speeds, scale, and scope.

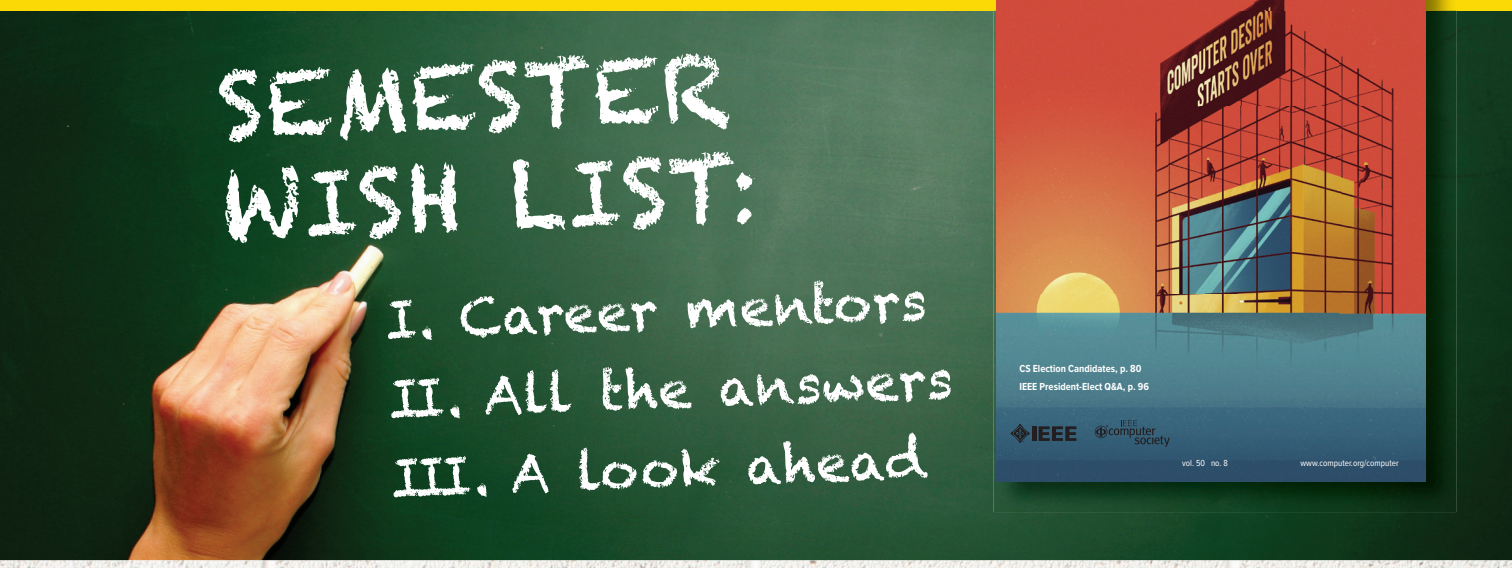
Both attack and defense will benefit from AI technologies, but I believe that AI has the capability to tip the scales more toward defense. There will be better offensive and defensive AI techniques. But here's the thing: defense is currently in a worse position than offense precisely because of the human components. Present-day attacks pit the relative advantages of computers and humans against the relative weaknesses of computers and humans. Computers moving into what are traditionally human areas will rebalance that equation.

Roy Amara famously said that we overestimate the short-term effects of new technologies, but underestimate their long-term effects. AI is notoriously hard to predict, so many of the details I speculate about are likely to be wrong—and AI is likely to introduce new asymmetries that we can't foresee. But AI is the most promising technology I've seen for bringing defense up to par with offense. For Internet security, that will change everything. ■

Bruce Schneier is a security technologist and a Fellow at the Berkman Klein Center for Internet and Society at Harvard University. He's also the chief technology officer of IBM Resilient and special advisor to IBM Security. Contact him via www.schneier.com.

Copublished by the IEEE Computer and Reliability Societies

IEEE computer society



SHARE THE GIFT OF KNOWLEDGE:

Give Your Favorite Student a Membership to the IEEE Computer Society!

With an **IEEE Computer Society Membership**, your student will be able to build their network, learn new skills, and access the best minds in computer science before they're even out of school. Your gift includes thousands of key resources that will quickly transition them from classroom to conference room, such as:

- **A subscription to *Computer* magazine** (12 issues per year)
- **A subscription to *ComputingEdge*** (12 issues per year)
- **Local chapter membership**
- **Full access to the Computer Society Digital Library**
- **Eligible for 3 student scholarships where we give away US$40,000 yearly**
- **Skillsoft:** Learn new skills anytime with access to 3,000 online courses, 11,000 training videos, and 6,500 technical books.
- **Books24x7:** On-demand access to 15,000 technical and business resources.
- **Unlimited access to computer.org and myCS**
- **Conference discounts**
- **Members-only webinars**
- **Deep member discounts** on programs, products, and services

Give Your Gift at: www.computer.org/2018gift