Cybersecurity Education and Formal Method

James H. Davenport[1] and Tom Crick[2]

[1] Department of Computer Science, University of Bath, UK
jhd@cs.bath.ac.uk
[2] Department of Computer Science, Swansea University, UK
thomas.crick@swansea.ac.uk

Abstract. Formal methods have been largely thought of in the context of safety-critical systems, where they have achieved major acceptance. Tens of millions of people trust their lives every day to such systems, based on formal proofs rather than "we haven't found a bug" (yet!); but why is "we haven't found a bug" an acceptable basis for systems trusted with hundreds of millions of people's personal data?
This paper looks at some of these issues in cybersecurity, primarily focused on the UK as a case study, and the extent to which formal methods, ranging from "fully verified" to better tool support, could help. More importantly, recent policy reports and curricula initiatives appear to recommended formal methods in the limited context of "safety critical applications"; we suggest this is too limited in scope and ambition. Not only are formal methods neded in cybersecurity, the repeated weaknesses of the cybersecurity industry provide a powerful motivation for formal methods.

Keywords: Formal methods, cybersecurity, curricula

1 Introduction

Formal methods, when they have been thought of at all, have been largely thought of in the context of safety-critical systems, where they have achieved major acceptance in what is, alas, an unsung area of sotware development. Tens of millions of people trust their lives every day to such systems, but nearly all are unaware of these systems, and the extent to which they are enormous successes. Even people "who ought to know better" don't. One of the authors quoted the Ligne 14 performance figures (software shipped in 1999 and no bugs reported [1]) to a major figure in the commercial software industry, to be told that he was lying, as this was utterly impossible.

Formal methods *ought* to be much more widely used in the cybersecurity industry. This is much more visible (because it has many conspicuous failures) than the largely invisible safety-critical industry. However, formal methods are not currently widely adopted here, and hence there is tremendous scope for growth and adoption of formal methods.

2 Cybersecurity

Cybersecurity[3] failures abound, and the number of people that can be affected by even a single failure is amazing — 148 million for Equifax [2] and probably more for the Starwood[4] breach: a number [3] "downgrades" to 383 million. The financial costs can be substantial: bankruptcy in the case of American Medical Collection Agency [4] and a provisional £183M fine for British Airways [5]. These problems have attracted attention at the highest scientific levels [6].

There are many reasons for cybersecurity failures, and even a given failure may have multiple causes. For example, the U.S. Government investigation [7] into Equifax states "Equifax's investigation of the breach identified four major factors including identification, detection, segmenting of access to databases, and data governance that allowed the attacker ...". However, none of these would have been triggered had it not been for the original bug in the Apache code [8], which was of the well-known (Number 1 Application Security Risk in [9]) family of "Injection" (or "Remote Code Execution") attacks, and which would probably have been detected by an automatic taint analysis tool such as [10].

Though attributing causes at scale is difficult, a well-known textbook [11] claims that about 50% of security breaches are caused by coding errors. Hence it behoves security practitioners to look seriously at coding errors, while recognising that this is only one facet of the problem. This is taken up by the Payments Card Industry in [12], essentially the only world-wide mandatory security standard, in two requirements.

6.5 Address common coding vulnerabilities in software-development processes as follows:

- Train developers at least annually in up-to-date secure coding techniques, including how to avoid common coding vulnerabilities;
- Develop applications based on secure coding guidelines.

6.6 For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods:

- Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes;
- Installing an automated technical solution that detects and prevents web-based attacks (for example, a web-application firewall) in front of public-facing web applications, to continually check all traffic.

[3] The precise definition of cybersecurity is debatable: we can take is as failures of security, generally defined as "preserving the CIA — Confidentiality, Integrity and Availability" of digital information, where computer system played a critical part in the failure.

[4] Generally called "Marriott", but in fact due to the Starwood chain before Marriott took it over.

It is noteworthy that, despite apparently insisting on secure coding in 6.5, they require the additional defences in 6.6, realising that *errare humanum est*, and the 6.5-developed code may not actually be secure. Is it possible (the author thinks so, but the experiment has yet to be performed) that adding formal methods to 6.5 would render 6.6 redundant, or at least mean that 6.6 should be restricted to finding design errors, rather than debugging 6.5 failures? Full formal verification of a complete system should certainly suffice.

> Complete formal verification is the only known way to guarantee that a system is free of programming errors. [13, describing seL4: a verified operating system]

Such a verified operating system has been used in medical devices, but probably not sufficiently widely, as 500,000 already-fitted pacemakers have had to be upgraded through security weaknesses [14], and insulin pumps are also vulnerable [15]. See [16] for a recent update on seL4. However, most of us do not have the opportunity to start from scratch, and have to live on top of imperfect, unverified systems, interoperating with other systems via large, generally unverified, protocols, such as TLS [17].

3 Agile versus Secure

"Agile Development" [18] is a major theme in software development. Mark Zuckerberg can be said to have taken this theme to the extreme in 2009.

> "Move fast and break things" is Mark's prime directive to his developers and team. "Unless you are breaking stuff," he says, "you are not moving fast enough." [19]

In both safety-critical and security-conscious programming, "breaking things" comes with a very high price. Aeroplanes can't be uncrashed, and data can't be unleaked.

The problems with using "Agile" methods in security are well-documented, at practitioner level, e.g. a recent "Security + Agile = FAIL" presentation [20], in many theoretical analyses as well as the interview-based research in [21] for small teams and [22] for large multi-team projects. Both mention team expertise in security as a significant problem.

From [21] The overall security in a project depends on the security expertise of the individuals, either on the customer or developer side. This corresponds to the agile value of "individuals and interaction over processes and tools" [18, Value 1].

From [22] The interviewees generally agree that more could be done to provide security education and training to employees. Without prompting, several interviewees mentioned training as an important factor for increasing security awareness and expertise.

It is very hard to take security seriously in this setting.

From [21] security "is only of interest [to the customer] when money-aspects are concerned".

From [22] One Test Manager articulated his team view that "security is not currently seen as part of working software, it only costs extra time and it doesn't provide functionality". With less focus on providing extensive (security) documentation typical for agile, ineffective knowledge sharing between security officers and agile team members is especially problematic.

From [23] (A more general survey, but many papers surveyed were "Agile") "Security is often referred to as a NFR [non-functional requirement] in that it is expected to be included as part of high quality code development, but is rarely listed as an explicit requirement. As a result, developers prioritise security below more-visible functional requirements or even easy-to-measure activities such as closing bug tracking tickets."

It would be tempting to conclude that "Agile" and "Secure" are, or at least are close to being, mutually contradictory. But there has been some analysis of the same apparent contradiction in the safety-critical industry [24]. Other than "Embedded Systems"[5] [24, §3.6], this analysis of the problems is fairly close to the practitioner view in [20], and we could reasonably ask what lessons could be carried across.

4 The Need for Tools

There are two key points.

From [24, §4.1] Strong static verification tools tend to complement (not replace) human-driven review[6]. The tools are very good at some problems (e.g. global data flow analysis, theorem proving) where humans are hopeless, and vice versa. If we do the static verification first, then we can adjust manual review processes and check-lists to take advantage of this.

From [24, §6] The sixty-four-million-dollar-question, it seems, is how much "up-front" work is "just right" for a particular project. We doubt there's a one-size-fits-all approach, but surely the answer should be informed by disciplined requirements engineering of non-functional properties (e.g. safety, security and others) that can inform the design of a suitable architecture and its accompanying satisfaction argument.

Facebook grew, security (and "product quality" in general: it is not clear whether security was the main driver here) became more important, and by 2014 Zuckerberg had changed his views.

> "Move fast with stable infrastructure." It "may not be quite as catchy as 'move fast and break things,"' Zuckerberg said with a smirk. "But it's how we operate now." [27]

[5] Actually, Embedded Systems are a comparatively neglected, but important, cybersecurity area. See, for example, [25] for a description of a pervasive design fault in the "home security" market.

[6] A point made in the context of XP and Agile in 2004 [26].

One might think his views were converging with the views of [24]. However, the Heartbleed story should remind us that the fact that a modification "has no new security considerations" *as designed* [28] doesn't mean that an implementation of that idea has no new security considerations. Hence the call in [24, §4.1] for strong static verification tools. Such tools are generally seen as expensive and slowing down the development process, but [29] shows that they need not be. In particular, they show that, for a real application (890,000 physical lines of Ada code), the cost of incremental verification can be reduced from "nightly" to "coffee", and hence can reasonably form part of a continuous integration toolchain, as is done at the company studied in [29]. Readers might comment that their own applications are not in Ada, but [30, §5.6] discusses mixed-language programming, especially with C. A similar point is made in [31], describing the Infer tool running on Java/Objective C/C++, where moving from overnight reporting to near real-time reporting moved the fix rate from 0% to 70%.

That these techniques are reaching the mainstream of cybersecurity can be seen from Amazon Web Services adoption of them [32], Google [33], Facebook [31], and the recent DefectDojo release by OWASP [34].

5 The Scope of Tools and Formal Methods

There is a substantial range of tools, and degrees of formality, and [24, §6] is probably correct in saying "We doubt there's a one-size-fits-all approach". At one extreme, there are the humble, but still surprisingly effective, `lint` and its equivalents, looking, essentially, for dangerous or dubious, though legal, syntax.

5.1 Ada and SPARK

At the other extreme, there are languages, such as the SPARK Ada subset [30] designed with verification in mind and heavily employed in the safety-critical sector such as railways and air traffic control, which can also be deployed for demanding secure applications, such as an RFC4108-compliant secure download system for embedded systems [35].

5.2 C/C++

There is, however, a large middle ground between these two extremes. Even if the application is required to be in C or C++, there is a lot to be said for sticking to a safer (even if not provably safe) subset of the language *and associated libraries*, such as eschewing `strcpy` in favour of `strncpy`. This can often be enforced by static verification tools. We note that Google's "Zero Day" project reports [36] that 68% of all such zero-day exploits (i.e. exploits discovered in the wild first) were caused by memory corruption errors, and Microsoft report a very similar story [37].

There is a good survey of such subsets and standards in [38, Appendix F]. As that notes, the ISO standard for secure C coding [39] has the unusual (for

this middle ground) but important concept of "taint analysis" (as in [10]): input data should be considered "tainted" until it has been sanitised. This is particularly important for network-oriented applications, where it is natural for the programmer to believe that the other party is behaving correctly (see **Heartbleed** above).

5.3 Java

Closer to the SPARK Ada end of the spectrum we find Safety-Critical Java [40]. The authors do not have enough experience with this to comment directly. However, the Java ecosystem (Stack Overflow etc.) is far from security-aware [41]. The fact that an application is in Java doesn't mean it's free from security coding errors: see [42] for a recent example.

There is a static analysis security tool for Java described in [10]. As with [39], this has "taint analysis" as its major feature, and at the time it spotted some significant-seeming problems.

5.4 JavaScript

JavaScript is a particular problem for Security. There are some verification tools, e.g. GATEKEEPER as described in [43]. However, even if it were possible to guarantee a particular piece of stand-alone JavaScript, that is not how the current paradigm operates. As [44] writes:

> Much of the power of modern Web comes from the ability of a Web page to combine content and JavaScript code from disparate servers on the same page. While the ability to create such mash-ups is attractive for both the user and the developer because of extra functionality, code inclusion effectively opens the hosting site up for attacks and poor programming practices within every JavaScript library or API it chooses to use.

Though not explicit in this statement, an additional weakness is that this combination is *dynamic*. The obvious solution would be some kind of sandboxing of the external resources relied upon, but the nature of JavaScript makes this difficult. [45] describe one such sandboxing, but it only works for a subset of JavaScript and relies on a combination of filtering, rewriting and wrapping to guarantee security. That it can do so at all is a remarkable feat of formal methods, given that previous attempts such as Facebook's FBJS have subtle flaws [46], and that the formal semantics of JavaScript being relied upon are very much a piece of reverse engineering.

In fact the dynamic loading from multiple sites is often not good for performance, and web performance engineers recommend tools to bundle the pages: this could usefully be combined with the sort of protection described by [45].

An alternative solution is suggested by Google,who are introducing a form of taint analysis into Chrome [47] through run-time typing. When enabled, this

means that the 60+ dangerous DOM API functions can only be called with arguments whose type is that emitted by `TrustedTypes` functions. Google expects that these functions would be manually verified, but this does open the door to formal verification of *certain* security policies in what is currently a very challenging environment for formal methods. However, these checks can be easily fudged, and the authors foresee examples of this on StackOverflow analogous to the `csrf().disable()` "suggestion" described below in point 3.

6 Education

[12, Requirement 6.5] called for education of developers. Education of mainstream programmers, as opposed to cybersecurity specialists, in cybersecurity has been neglected until recently, and this neglect has been lamented as far as the Harvard Business Review [48]. Developments in professional accreditation are changing this [49]. However, there are limitations, even beyond *errare humanum est*, in relying on education.

1. There is experimental evidence that both trained students [50] and professional developers [51] will ignore security considerations unless *explicitly* instructed to take them into account. Lest this be thought to be a purely academic exercise with little relevance to the real world, consider the recent ¥55M password problem described in [52].
2. There is field evidence that explicit requirements such as [12] are ignored in practice, e.g. the Forever 21 breach [53], or Macy's [54]. They may also not be communicated down the software supply chain, as in the Ticketmaster case [55].
3. Many educational resources, both formal textbooks [56] and informal resources such as Stack Overflow [57], pay very little attention to security, and indeed can be positively harmful. The discussion in Stack Overflow (analysed in [41, §4.3.1]) of cross-site request forgery (CSRF — this was in the OWASP top 10 in 2013 [58], but dropped from [9] "as many frameworks include CSRF defenses") is especially worrying. By default, Spring implicitly enables protection against this. But all the accepted answers to CSRF-related failures simply suggested disabling the check. There were no negative comments about this, and indeed a typical response is "Adding `csrf().disable()` solved the issue!!! I have no idea why it was enabled by default".

As we have noted, [12] both mandates education and does not rely solely on it.

However, as the safety-critical community laments (at least in the U.K. and U.S.A.: cultures do differ here), there is very little training in formal methods for most undergraduates.

7 Conclusions

As the media never tire of saying, there are far too many security breaches, and, though they have multiple causes, [11] claims that about 50% of security

breaches are caused by coding errors. There appears to be a culture of accepting these, with the U.S. Government investigation [7] into Equifax blaming many factors but not the actual bug, and [12] taking a "necessary but not sufficient" approach to education in secure coding.

Education Could certainly do better [48], though there are encouraging signs [49] and useful ideas when it comes to improving informal resources [59]. However, informal resources can be dangerous when it comes to security, and [49] recommends giving *all* students the advice in [60]: "If you pick up a SSL/TLS answer from Stack Overflow, there's a 70% chance it's insecure". More training in formal methods would be welcomed, at least in those cultures where it is lacking.

Customers/Managers need to be much more upfront about security requirements [50, 51], and enforce (e.g. by requiring tool support during any CI/CD process, such as [29] describe) at least "middle ground" requirements. In the case of outsourced development, explicit penalty clauses for failing penetration tests should concentrate the developers' minds.

C/C++ people These programmers should be much more aware of techniques for secure coding, such as those described in [38, Appendix F], and the various tools for static analysis.

Java people In view of the significance of injection attacks (Number 1 in [9]), programmers should be aware of taint analysis, as in [10].

JavaScript people There are some techniques, such as [45], for protecting JavaScript applications, but they are not deployable in the typical JavaScript "dynamic loading web page" environment. Furthermore this environment is basically antithetical to security, as British Airways is learning to the cost of £183M [5].

1) Hence the first real challenge of JavaScript lies with the tool makers: there are, as far as the author knows, no JavaScript verifiers in existence, and no page-bundler that checks for version drift, or does incremental verification (which might be comparatively cheap, as in [29]).

2) An alternative approach might be to change the JavaScript model. This is advocated in [61], based on their analysis of what third-party scripts do in the wild. This is not a completely radical idea: Google is testing its `TrustedTypes` feature [47], with the motivation "The DOM API is insecure by default and requires special treatment to prevent XSS".

Empirical Research There is not much analysis of the efficacy of various techniques in security programming. [62] compares various techniques, and states the following.

> Based on our case study [of two large programs], the most efficient vulnerability discovery technique is automated penetration testing. Static analysis finds more vulnerabilities but the time it takes to classify false positives makes it less efficient than automated testing.

This assumes that "false positives" are acceptable, a debatable point of view. It would be good to have more such research.

Tool developers There is a lack of tools (or at least a lack of awareness of tools) that can be neatly integrated into a security programming toolchain the way such tools are integrated in safety-critical toolchains [29].

Acknowledgements

This paper is adapted from a talk given at the 2019 Working Formal Methods Symposium (FROM2019) in Timisoara, Romania. The first author is grateful to the Fulbright Programme for a Cybersecurity Scholarship at New York University in 2017, and to many correspondents and discussions, notably Alastair Irons, Tom Prickett and Tim French.

References

1. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Verifying b proof rules using deep embedding and automated theorem proving. In: International Conference on Software Engineering and Formal Methods, Springer (2011) 253–268
2. Bloomberg: Equifax Hack Lasted for 76 Days, Compromised 148 Million People, Government Report Says. `http://fortune.com/2018/12/10/equifax-hack-lasted-for-76-days-compromised-148-million-people-government-report-says/` (2018)
3. Irwin, L.: Marriott downgrades severity of 2018 data breach: 383 million customers affected. `https://www.itgovernance.co.uk/blog/marriott-downgrades-severity-of-2018-data-breach-383-million-customers-affected` (2019)
4. Ford, N.: Medical debt collection agency files for bankruptcy protection after data breach. `https://www.itgovernance.co.uk/blog/medical-debt-collection-agency-files-for-bankruptcy-protection-after-data-breach` (2019)
5. The Guardian: BA faces £183m fine over passenger data breach. `https://www.theguardian.com/business/2019/jul/08/ba-fine-customer-data-breach-british-airways` (2019)
6. Royal Society: Progress and research in cybersecurity: Supporting a resilient and trustworthy system for the UK. `http://royalsociety.org/cybersecurity` (2016)
7. United States Government Accountability Office: Actions Taken by Equifax and Federal Agencies in Response to the 2017 Breach. `https://www.gao.gov/assets/700/694158.pdf` (2018)
8. Lenart, L.: Security Bulletin S2-045. `https://cwiki.apache.org/confluence/display/WW/S2-045` (2017)
9. Open Web Application Security Project (OWASP): The Ten Most Critical Web Application Security Risks. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project` (2017)
10. Livshits, V., Lam, M.: Finding Security Vulnerabilities in Java Applications with Static Analysis. In: Proceedings USENIX Security Symposium. (2005) 271–286
11. McGraw, G.: Software Security — Building Security In. Addison-Wesley (2006)
12. Payment Card Industry Security Standards Council (PCI SSC): Requirements and Security Assessment Procedures Version 3.2.1. `https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf` (2018)

13. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T.: seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (2009) 207–220
14. The Guardian: Hacking risk leads to recall of 500,000 pacemakers due to patient death fears. https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update (2017)
15. Newman, L.: Hackers Made an App That Kills to Prove a Point. https://www.wired.com/story/medtronic-insulin-pump-hack-app (2019)
16. Heiser, G.: What's new in the world of seL4. https://archive.fosdem.org/2019/schedule/event/world_of_sel4/ (2019)
17. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC-8446 (2018)
18. Beck,K. *et al.*: The Agile Manifesto. http://agilemanifesto.org/ (2001)
19. Blodget, H.: Mark Zuckerberg On Innovation. https://www.businessinsider.com/mark-zuckerberg-innovation-2009-10 (2009)
20. Lane, A.: Security + Agile = FAIL. https://securosis.com/assets/library/presentations/Security/AgileFAIL_OWASP.ppt_.pdf (2018)
21. Bartsch, S.: Practitioners' Perspectives on Security in Agile Development. In International Conference on Availability Reliability and Security (2011) 479–484
22. van der Heijden, A., Broasca, C., Serebrenik, A.: An empirical perspective on security challenges in large-scale agile software development. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '18, New York, NY, USA, ACM (2018) 45:1–45:4
23. Tahaei, M., Vaniea, K.: A Survey on Developer-Centred Security. https://groups.inf.ed.ac.uk/tulips/papers/A_Survey_on_Developer_Centred_Security.pdf (2019)
24. Chapman, R.: Industrial experience with Agile in high-integrity software development. In Parsons, M., Anderson, T., eds.: Developing Safe Systems: Proceedings of the Twenty-fourth Safety-critical Systems Symposium, Safety-Critical Systems Club (2016) 143–154
25. O'Connor, T., Enck, W., Reaves, B.: Blinded and Confused: Uncovering Systemic Flaws in Device Telemetry for Smart-Home Internet of Things. In: Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks. (2019) 140–150
26. Wäyrynen, J., Bodén, M., Boström, G.: Security Engineering and eXtreme Programming: an Impossible Marriage? Extreme programming and agile methods-XP/Agile Universe (2004) 117–128
27. Statt, N.: Zuckerberg: 'Move fast and break things' isn't how Facebook operates anymore. https://www.cnet.com/news/zuckerberg-move-fast-and-break-things-isnt-how-we-operate-anymore/ (2014)
28. Seggelmann, R., Tuexen, M., Williams, M.: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. https://tools.ietf.org/html/rfc6520 (2012)
29. Brain, M., Schanda, F.: A lightweight technique for distributed and incremental verification. In Joshi, R., Müller, P., Podelski, A., eds.: Verified Software: Theories, Tools, Experiments. Volume 7152 of LNCS., Berlin–Heidelberg–New York, Springer (January 2012) 114–129
30. Chapman, R., Moy, Y.: AdaCore Technologies for Cyber Security. https://www.adacore.com/books/adacore-tech-for-cyber-security (2018)

31. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.: Scaling static analyses at Facebook. Communications of the ACM **62** (2019) 62–70
32. Vogels, W.: Proving security at scale with automated reasoning. https://www.allthingsdistributed.com/2019/05/proving-security-at-scale-with-automated-reasoning.html (2019)
33. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushion, L., Jaspan, C.: Lessons from building static analysis tools at Google. Commun. ACM **61**(4) (2018) 58–66
34. Open Web Application Security Project (OWASP): DefectDojo: OpenSource Application Security Management. https://www.defectdojo.org (2019)
35. Chapman, R.: Development and Formal Verification of Secure Updates for Embedded Systems (slides from Verification 2018). http://www.testandverification.com/conferences/verification-futures/vf2018/ (2018)
36. Google (Project Zero): 0day "In the Wild". https://googleprojectzero.blogspot.com/p/0day.html (2019)
37. Thomas, G.: A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (2019)
38. Centre for the Protection of National Infrastructure: Rail Code of Practice for Security-Informed Safety. CPNI (2019)
39. ISO/IEC: TS 17961:2013, Information technology — Programming languages, their environments & system software interfaces — C Secure Coding Rules. https://www.iso.org/standard/61134.html (2013)
40. Cavalcanti, A., Miyazawa, A., Wellings, A., Woodcock, J., Zhao, S.: Java in the Safety-Critical Domain. SETSS 2016: Engineering Trustworthy Software Systems (2017) 110–150
41. Meng, N., Nagy, S., Yao, D., Zhuang, W., Arango Argoty, G.: Secure coding practices in Java: Challenges and vulnerabilities. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE) (2018) 372–383
42. Google (Chris Povirk): Denial of Service vulnerability for servers that use Guava and deserialize attacker data. https://groups.google.com/forum/\#!topic/guava-announce/xqWALw4W1vs/discussion (2018)
43. Guarnieri, S., Livshits, B.: GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: USENIX Security Symposium. Volume 10. (2009) 76–85
44. Meyerovich, L., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In: 2010 IEEE Symposium on Security and Privacy, IEEE (2010) 481–496
45. Maffeis, S., Mitchell, J., Taly, A.: Isolating JavaScript with Filters, Rewriting, and Wrappers. In: Proceedings ESORICS 2009. (2009) 505–522
46. Maffeis, S., Taly, A.: Language-based isolation of untrusted Javascript. In: Proceedings 22nd IEEE Computer Security Foundations Symposium. (2009) 77–91
47. Kotowicz, K.: Trusted Types help prevent Cross-Site Scripting. https://developers.google.com/web/updates/2019/02/trusted-types (2019)
48. Cable, J.: Every Computer Science Degree Should Require a Course in Cybersecurity. https://hbr.org/2019/08/every-computer-science-degree-should-require-a-course-in-cybersecurity (2019)
49. Crick, T., Davenport, J., Irons, A., Prickett, T.: A UK Case Study on Cybersecurity Education and Accreditation. https://arxiv.org/abs/1906.09584 (2019)
50. Naiakshina, A., Danilova, A., Tiefenau, C., Smith, M.: Deception Task Design in Developer Password Studies: Exploring a Student Sample. In Fourteenth Sympo-

sium on Usable Privacy and Security (SOUPS 2018). USENIX Association (2018) 297–313
51. Naiakshina, A., Danilova, A., Gerlitz, E., von Zezschwitz, E., Smith, M.: " If you want, I can store the encrypted password": A Password-Storage Field Study with Freelance Developers. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, ACM (2019) 140:1–140:12
52. Cimpanu, C.: 7-Eleven Japanese customers lose $500,000 due to mobile app flaw. https://www.zdnet.com/article/7-eleven-japanese-customers-lose-500000-due-to-mobile-app-flaw/ (2019)
53. Biscoe, C.: MyFitnessPal data breach: 150 million app users affected. https://www.itgovernance.co.uk/blog/myfitnesspal-data-breach-150-million-app-users-affected/ (2018)
54. Blackmon, A.: Macy's hit by data breach. https://eu.freep.com/story/money/business/2018/07/06/macys-data-breach-online/763074002/ (2018)
55. Inbenta (CEO): Inbenta and the Ticketmaster Data Breach. http://web.archive.org/web/20181121184620/ (2018)
56. Taylor, C., Sakharkar, S.: ');DROP TABLE textbooks;–: An Argument for SQL Injection Coverage in Database Textbooks. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). ACM (2019) 191–197
57. Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., Fahl, S.: Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. 38th IEEE Symposium on Security and Privacy (SP) (2017) 121–136
58. Open Web Application Security Project (OWASP): The Ten Most Critical Web Application Security Risks. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf (2013)
59. Fischer, F., Xiao, H., Kao, C.Y., Stachelscheid, Y., Johnson, B., Razar, D., Fawkesley, P., Buckley, N., Böttinger, K., Muntean, P., Grossklags, J.: Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography. 28th USENIX Security Symposium (USENIX Security 19) (2019)
60. Chen, M., Fischer, F., Meng, N., Wang, X., Grossklags, J.: How Reliable is the Crowdsourced Knowledge of Security Implementation? https://arxiv.org/abs/1901.01327 (2019)
61. Zhang, M., Meng, W., Lee, S., Lee, B., Xing, X.: All Your Clicks Belong to Me: Investigating Click Interception on the Web. https://www.microsoft.com/en-us/research/uploads/prod/2019/03/zhang-observer.pdf (2019)
62. Austin, A., Williams, L.: One technique is not enough: A comparison of vulnerability discovery techniques. In: Proceedings 2011 International Symposium on Empirical Software Engineering and Measurement. (2011) 97–106