Formal Methods for Security Functionality and for Secure Functionality

Erik Poll

Digital Security group, Radboud University Nijmegen, The Netherlands
erikpoll@cs.ru.nl

With cyber security becoming a growing concern, it has naturally attracted the attention of researchers in formal methods. One recent success story here is TLS: the development of the new TLS 1.3 specification has gone hand-in-hand with efforts to verify security properties of formal models [5] and the development of a fully verified implementation [3]. Earlier well-known success stories in using formal methods for security are the verifications of operating system kernels or hypervisors, namely seL4 [7] and Microsoft's Hyper-V [10].

These examples – security protocols and OS kernels – are applications whose primary purpose is to provide security. It is natural to apply formal methods to such systems: they are by their very nature security-critical and they provide some security functionality that we can try to specify and verify.

However, we want *all* our systems to be secure, not just these security systems. There is an important difference between *secure* functionality and *security* functionality, or – given that most functionality and most security problems are down to software – between *software security* and *security software* [11]. Many, if not most, security problems arise in systems that have no specific security objective, say PDF viewers or video players, but which can still be hacked to provide attackers with unwanted functionality they can abuse.

Using formal methods to prove security is probably not on the cards of something as complex as a PDF viewer or video player. Just defining what it would mean for such a system to be secure is probably already infeasible. Still, formal methods can be useful, to prove the absence of certain types of security flaws or simply find security flaws. Successes here have been in the use of static analysis in source code analysers, e.g. tools like Fortify SCA that look for flaws in web applications and tools like Coverity that look for memory vulnerabilities in C(++) code. Another successful application of formal methods is the use of symbolic (or concolic) execution to generate test cases for security testing, as in SAGE [6] or, going one step further, not just automatically finding flaws but also automatically generating exploits, as in angr [16].

Downside of these approaches is that they are post-hoc and can only look for flaws in existing code. The *LangSec* paradigm [9, 4], on the other hand, provides ideas on how to prevent many security problems *by construction.* Key insights are that most security flaws occur in input handling and that there are several root causes in play here. Firstly, the input languages involved (e.g. file formats and network protocols) are complex, very expressive, and poorly, informally, specified. Secondly, there are *many* of these input languages, sometimes nested or stacked. Finally, parsers for these languages are typically hand-written, with parsing code scattered throughout the application code in so-called shotgun parsers [12]. With clearer, formal specifications of input languages and generated parser code much security misery could be avoided. (Recent initiatives in tools for parser generation here include Hammer [1] and Nail [2].) Given that formal languages and parser generation are some of the most basic and established formal methods around, it is a bit of an embarrassment to us as formal methods community that sloppy language specifications and hand-coded parsers should cause so many security problems.

Some security flaws in input handling are not so much caused by *buggy* parsing of inputs, but rather by the *unexpected* parsing of input [13]. Classic examples of this are command injection, SQL injection, and Cross-Site Scripting (XSS). Tell-tale sign that unwanted parsing of input may be happening in unexpected places is the heavy use of strings as data types [14].

Information or data flow analysis can be used to detect such flaws; indeed, this is a standard technique used in the source code analysis tools mentioned above. These flaws can also be prevented by construction, namely by using type systems. A recent example of this is the Trusted Types browser API [8] by Google, where different types are used to track different kinds of data and different trust level of data to prevent XSS vulnerabilities, esp. the DOM-based XSS vulnerabilities that have proved so difficult to root out.

To conclude, formal methods cannot only be used to *prove* security of security-critical applications and components – i.e. the security software –, but they can be much more widely used to *improve* security by ruling out of the root causes behind security flaws in input handling, and do so by construction, and hence improve software security in general. Moreover, some very basic and lightweight formal methods can be used for this: methods that we teach – or should be teaching – our students in the first years of their Bachelor degree, such as regular expressions, finite state machines, grammars, and types. Indeed, in my own research I have been surprised to see how useful the simple notion of finite state machine for describing input sequences is to discover security flaws [15].

That we have not been able to get these basic techniques into common use does not say much for our success in transferring formal methods to software engineering practice. Still, looking at the bright side, it does suggest opportunities for improvement.

References

1. Anantharaman, P., Millian, M.C., Bratus, S., Patterson, M.L.: Input handling done right: Building hardened parsers using language- theoretic security. In: Cybersecurity Development (SecDev). pp. 4–5. IEEE (2017)
2. Bangert, J., Zeldovich, N.: Nail: A practical tool for parsing and generating data formats. In: OSDI'14. pp. 615–628. Usenix (2014)
3. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified models and reference implementations for the TLS 1.3 standard candidate. In: Security and Privacy (S&P'17). pp. 483–502. IEEE (2017)

4. Bratus, S., Locasto, M.E., Patterson, M.L., Sassaman, L., Shubina, A.: Exploit programming: From buffer overflows to weird machines and theory of computation. ;login: pp. 13–21 (2011)
5. Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T.: A comprehensive symbolic analysis of TLS 1.3. In: SIGSAC Conference on Computer and Communications Security (CCS'17). pp. 1773–1788. ACM (2017)
6. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (Mar 2012)
7. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: ACM SIGOPS. pp. 207–220. ACM (2009)
8. Kotowicz, K.: Trusted types help prevent cross-site scripting (2019), blog, https://developers.google.com/web/updates/2019/02/trusted-types
9. LangSec: Recognition, validation, and compositional correctness for real world security (2013), uSENIX Security BoF hand-out. Available from http://langsec.org/bof-handout.pdf
10. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V hypervisor with VCC. In: International Symposium on Formal Methods. pp. 806–809. Springer (2009)
11. McGraw, G.: Software security. IEEE Security & Privacy **2**(2), 80–83 (2004)
12. Momot, F., Bratus, S., Hallberg, S.M., Patterson, M.L.: The seven turrets of Babel: A taxonomy of LangSec errors and how to expunge them. In: Cybersecurity Development (SecDev'16). pp. 45–52. IEEE (2016)
13. Poll, E.: LangSec revisited: input security flaws of the second kind. In: Workshop on Language-Theoretic Security (LangSec'18). IEEE (2018)
14. Poll, E.: Strings considered harmful. ;login: **43**(4), 21–26 (2018)
15. Poll, E., de Ruiter, J., Schubert, A.: Protocol state machines and session languages: specification, implementation, and security flaws. In: Workshop on Language-Theoretic Security (LangSec'15). pp. 125 – 133. IEEE (2015)
16. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK:(state of) the art of war: Offensive techniques in binary analysis. In: Symposium on Security and Privacy (SP'16). pp. 138–157. IEEE (2016)