Formal Methods for Security?

Erik Poll
Digital Security group
Radboud University
erikpoll@cs.ru.nl

This position paper sketches some opportunities in applying formal methods for security, more specifically security of software.

A killer application for formal methods?

At first sight – and even at second sight – security looks like an interesting application area for formal methods.

One reason for this is that security flaws have a higher impact than more harmless bugs. This might justify or even require the extra effort to invest in using formal methods. Indeed, the highest levels of certification using the Common Criteria security evaluation standard require the use of formal methods.

Another reason is that some security problems are orthogonal to – or at least largely independent of – the functionality of a system. Indeed, security vulnerabilities can be seen as 'anti-functionality': functionality that is unintentionally provided, also to attackers, which should not be available at all. Given that writing complete functional specifications is hard, and totally unfeasible in most cases, concentrating on partial specs that ensure some generic safety properties problems might provide a better return of investment, also because such specs could be re-used across applications.

Unfortunately, things are not so simple. Security properties can be tricky to specify. Indeed, attackers can be very creative in finding and exploiting new loopholes. A common and natural way to specify security properties is in a 'negative' way, by saying that something – some type of attack – is not supposed to be possible. For example, a web application should not be vulnerable to SQL injection or XSS. List of these negative properties are useful in testing, as they suggest negative tests (i.e. test-cases which are supposed to fail, by triggering some error response), but they are not immediately helpful in construction. Moreover, these lists of negative properties are typically incomplete. They only address a limited set of known potential problems, not all potential problems.

Moreover, sometimes security problems arise because it turns out fundamental assumptions about programs can be broken by an attacker, invalidating the very abstractions that we use in formal methods to reason about programs. Classic examples here are fault attacks (for example the Rowhammer attack to flip some bits in DRAM memory) or information leaking through side-channels (for example the Spectre and Meltdown timing attacks on modern CPUs). Of course, our formal models could be refined to accomodate these low-level attacks, but at the (high) cost of extra complexity.

Security functionality ≠ secure functionality

When investigating at the security of software, it is natural to focus on the security functionality, i.e. the functionality specifically intended to provide some security guarantees, such as access control checks or security protocols like SSL/TLS. Such functionality is obviously security-critical.

However, while this may suggest rewarding aspects or components to investigate, it is dangerous to fall into the trap of thinking that such security functionality is the only or even the most important area to look for problems. Not only the security functionality has to be secure: *all* functionality needs to be, as security vulnerabilities can lurk in any line of code that can be triggered by input controlled by the attacker. The bulk of security bugs is not in code specifically aimed at achieving some security goal, but is in more mundane functionality, say the parsing of PDF files or the rendering of some graphical format, with Flash as the most notorious example.

LangSec: back to basics?

The paradigm of LangSec (language-theoretic security)[1] provides very good insights into the root causes of the overwhelming majority of security flaws, namely bugs in handling input, which typically boil down to bugs in the parsing and processing input languages and formats, rather than bugs in the application logic.

One problem is with the input languages themselves: they are typically overly complex, too expressive, and poorly – and informally – specified. To make matters worse, there are many of these languages, at every level of the network and software stack, and they can be combined or nested. A further problem here is that the code to process these languages is typically hand-coded, and not obtained using parser generation.

Ironically – or embarrassingly, for the computing science community – theories for formal language definitions and parser generation are some of the oldest and most established areas in formal methods. Still, somehow the whole world is still writing long prose documents to specify languages and

A Research Agenda for Formal Methods in the Netherlands, September 3–4, Lorentz Center, Leiden, The Netherlands
2018.

[1]See also http:\langsec.org, esp. [1], or [7] for a more recent entry point into the LangSec literature.

protocols, and then hand-coding parsers – often in memory unsafe languages like C or C++, where the potential security impact of flaws is the biggest (namely remote code execution). There is a huge opportunity here to provide better notations and tools to prevent all this misery. Or maybe these already exist, and we should do a better job in training people – incl. our students – on how to use them?

One step further from formal specs and associated code generation for parsers and pretty printers would be domain-specific programming languages to support different input formats and languages as first class citizens, as envisioned in Wyvern [8].

Security Testing & Model Extraction

The past decade has seen a lot of fruitful interaction between formal methods and testing, also in security testing. An interesting trend is the use of formal methods, notably symbolic/concolic execution, for security testing [6, 11] or even going one step further and actually develop exploits as in the angr tool [10]. If we cannot get developers to use formal methods, maybe we should concentrate efforts on getting security testers and hackers to use formal methods? (Of course, with more robust parser code that has generated from formal language specs, as we argue for above, it should be harder to find security flaws ...)

Test techniques can also provide a way to obtain formal specs from implementations. Given the difficulty of obtaining formal models this is an interesting direction of work. Existing fuzzers can already reverse-engineer input formats [2, 3], and state machine inference can be used to extract security-relevant behaviour from code [9].

All such formal techniques for security testing or model inference could be combined with machine-based learning or AI approaches, to improve results and/or the level of automation.

Practical information flow

Information flow properties are an interesting class of security properties. Information flow can be used to track potential leakage of confidential information or to track the flow of tainted input to places where such input may do damage. Research on information flow has a long history, dating back to the 1970s [4], and ad-hoc information analyses are implemented in code analysis for security flaws – aka Static Application Security Testing (SAST), by tools such as Coverity, Checkmarx, or Fortify, but flexible and practical approaches to express and enforce information flow for programs in popular programming languages (e.g. [5] for Java) are still rare and not commonly used.

New (and safer) programming languages, new opportunities?

One positive development for security in recent years has been the advent of new programming languages – Rust, D, Go, Swift, Nim, ... – where safety is very much a design goal. Some of these languages are specifically aimed for low-level programming and might become viable, widely-used, and safer alternatives for C/C++ and then reduce the prevalence of memory corruption problems.

The advent of these new languages is a double-edged sword. An advantage is that they are designed to be more amenable to formal analysis and have some security guarantees built in at the language level. A downside is that new languages require new tools. Building and maintaining good formal methods tools is a major bottleneck, so here the advent of new languages is bad news. For researchers these new languages represent new research opportunities. This may be good news, if this new research gets us further, or bad news, if this research is merely repeating and recycling the same old ideas without getting us further.

References

[1] 2013. LangSec: Recognition, Validation, and Compositional Correctness for Real World Security. (2013). USENIX Security BoF hand-out. Available from http://langsec.org/bof-handout.pdf.

[2] J. Caballero, H. Yin, Z. Liang, and D. Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS'07*. ACM, 317–329.

[3] P.M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. 2009. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 110–125.

[4] D.E. Denning and P.J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513.

[5] W. Dietl, S. Dietzel, M.D. Ernst, K. Muşlu, and T.W. Schiller. 2011. Building and Using Pluggable Type-checkers. In *ICSE'11*. ACM, 681–690.

[6] P. Godefroid, M.Y. Levin, and D. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20.

[7] F. Momot, S. Bratus, S.M. Hallberg, and M.L. Patterson. 2016. The seven turrets of Babel: A taxonomy of LangSec errors and how to expunge them. In *Cybersecurity Development (SecDev)*. IEEE, 45–52.

[8] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. 2014. Safely composable type-specific languages. In *ECOOP'14 (LNCS)*, Vol. 8586. Springer, 105–130.

[9] E. Poll, J. de Ruiter, and A. Schubert. 2015. Protocol state machines and session languages: specification, implementation, and security flaws. In *Workshop on Language-Theoretic Security (LangSec'15), Symposium on Security and Privacy Workshops*. IEEE, 125 – 133.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK:(state of) the art of war: Offensive techniques in binary analysis. In *Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS'16*, Vol. 16. Internet Society, 1–16.