Guarantees from Dynamic Analysis

(Keynote at ICPC 2015)

Andreas Zeller
Computer Science, Saarland University, Saarbrücken, Germany
Email: zeller@acm.org

***Abstract*—Modern test generation techniques allow to generate as many executions as needed; combined with dynamic analysis, they allow for understanding program behavior in situations where static analysis is challenged or impossible. However, all these dynamic techniques would still suffer from the incompleteness of testing: If some behavior has not been observed so far, there is no guarantee that it may not occur in the future. In this talk, I introduce a method called *Test Complement Exclusion* that combines test generation and sandboxing to provide such a guarantee. Test Complement Exclusion will have significant impact in the security domain, as it effectively detects and protects against unexpected changes of program behavior; however, guarantees would also strengthen findings in dynamic software comprehension. First experiments on real-world Android programs demonstrate the feasibility of the approach.**

I. Introduction

Understanding what a program does or can do is one of the classic challenges of computer science. In the past, the usual setting was that the *producer* of a program needed to understand what was going on—for purposes such as reengineering, refactoring, extending, or debugging. This setting poses sufficiently many challenges on its own; however, comprehension would normally be facilitated by the cooperation of the program owner—in particular, by making source code or documentation available, and assuming that the program would not use intentional techniques that would make analysis hard or even impossible. If the program is *adverse*, though, and even actively prevents analysis, the challenge is much larger.

II. The Challenge of Program Analysis

Generally, program analysis falls into two categories: *static* analysis of program code and *dynamic* analysis of executions.

- **Static code analysis** sets an *upper bound* to what a program can do: If static analysis determines some behavior is impossible, it can be safely excluded. The challenge of *static analysis* is *overapproximation:* The analysis must often assume that more behaviors are possible than actually would be, and is challenged by code that is decrypted, interpreted, or downloaded at runtime only—limitations that can thus easily be exploited by anyone who wants to conceal (malicious) program behavior.
- **Dynamic runtime analysis** is the alternative to static analysis; it works on actual *executions,* and thus is not limited by code properties. In terms of program behavior, it sets a *lower bound:* Any (benign) behavior seen in past executions should be allowed in the future, too. The fundamental problem of dynamic analysis is *incompleteness:* If some behavior has not been observed so far, there is *no guarantee that it may not occur in the future.* Malware writers can easily exploit this incompleteness by making malicious behavior *latent:* For instance, one would activate malicious behavior only after some time, or in a specific (targeted) network, or when no dynamic analysis tool is run, each of which would defeat observation during testing.

When analyzing *cooperative* programs, overapproximation and incompleteness can both be overcome in practice by reasonable heuristics. But as it comes to *adverse* programs, these heuristics no longer apply; as researchers, we are empty-handed. Worse even, we are not talking about temporary deficits of our tools, but about *fundamental limitations* that malware writers happily exploit.

Analyzing adverse programs faces fundamental limitations.

III. Test Generation to the Rescue?

In a world where code would be hidden on dozens of servers and services, static analysis will be limited to local guarantees. As it comes to *global* behavior, *testing is the only option.* The good news is that advances in *test generation tools* now allow to generate as many executions as needed.

For software assessment, test generators are now able to work on a massive scale. On the Android platform, the popular MONKEY fuzz tester generates random streams of user events such as clicks, touches, or gestures for arbitrary Android programs. Frameworks like PUMA [1] have run dynamic analysis on 3,600 apps from the Google Play store. As any testing tool, these would not be limited by issues of scale or code not being accessible for analysis. However, they still share the fundamental incompleteness of execution analysis—and again, adverse programs can and will easily exploit this.

For complex programs,
testing might be the only means for analysis.

IV. Overcoming Incompleteness

What we need is a way to turn the incompleteness of dynamic analysis into a *guarantee*—similar to the guarantees provided by static and symbolic analysis. To this end, I suggest to combine two techniques, namely *test generation* and *enforcement*, in a principle called *test complement exclusion*—effectively *disallowing behavior not seen during testing:*

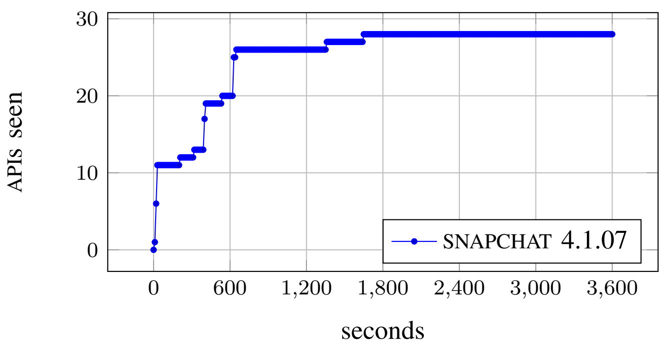

Fig. 1. DROIDMATE per-app API saturation. After 30 minutes (1,800 seconds), DROIDMATE has discovered 28 sensitive APIs used by SNAPCHAT.

- **Mining.** In the first phase, we *mine* program behavior. We use an *automatic test generator* to systematically explore program behavior, monitoring externally visible behavior such as program input, output, or resource accesses, and derive *abstract rules* from these. These rules *characterize the observed and expected program behavior,* a behavior to be enforced in production later.
- **Sandboxing.** In the second phase, we assume that *behavior not seen during testing should not take place during production either.* Consequently, if the program (unexpectedly) requires access to a new resource, the runtime system can prohibit access, or put the request on hold until the user explicitly allows it.

Here is how test complement exclusion in practice, taken from a set of initial experiments on Android apps [2]:

- To explore app behavior, we have built a GUI test generator named DROIDMATE, specifically designed to focus on those behaviors easily accessible from the GUI.
- To enforce behavior, we use a sandboxing tool named APPGUARD [3], allowing us to specifically block access to individual resources and APIs by means of *inline reference monitoring* [4].
- As behavior to be enforced, we monitor *access to sensitive resources,* specifically the APIs listed in [5].

The key question is: *Would simple, systematic GUI testing be able to cover all sensitive API resources such that a later sandbox would accurately flag unexpected resource accesses?*

Figure 1 shows how DROIDMATE explores the behavior of the popular SNAPCHAT picture messaging application: After 30 minutes, it has discovered 28 sensitive different APIs accessed by SNAPCHAT. Testing further even for several hours would not reveal more APIs; plus, the APIs discovered upfront (accessing the internet, the camera, or the microphone) would be used far more frequently than the last APIs discovered (opening support pages and references). Feeding this list into APPGUARD to enforce that SNAPCHAT cannot use any other API, we would protect against latent malicious behavior as well as someone compromising the SNAPCHAT binary. Should we have missed additional SNAPCHAT behavior during testing, the user would have to confirm a specific resource access; but for one, we would not know which API this would be; and second, we would assume this happens only on rarely used functions, as the most frequently used functions would be easily discoverable for users and test generators alike.

Excluding behavior not seen during testing turns the incompleteness of testing into a guarantee.

V. Conclusion and Consequences

Testing can do more; comprehension can do more; security can do more—but if we bring these together, we can get tremendous synergies. In the security domain, constraining an application to the behavior seen during testing protects against *latent malware,* as the malicious behavior cannot be significantly different from the benign behavior seen during testing. It could mitigate *vulnerabilities* where a program is compromised by an attacker to do something unexpected. And it would protect against *backdoors,* as by construction, these normally would not be discovered during regular testing.

Test complement exclusion could protect against latent malware, vulnerabilities, and backdoors.

On a more general note, test complement exclusion would help overcoming the fundamental limitation of dynamic analysis: Applied to program comprehension, it could ensure that all behavior seen indeed is all there can be, and provide all findings with guarantees. This does not come without challenges, though, notably when generating tests and finding suitable abstractions. However, the potential benefits may justify the effort—on a very large scale.

Testing, comprehension, security—better together!

Acknowledgments. My sincere thanks go to Michael Backes, Konrad Jamrozik, and Philipp von Styp-Rekowsky for their significant contributions. This work was funded by an European Research Council (ERC) Advanced Grant “SPECMATE – Specification Mining and Testing”.

References

[1] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, “PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps,” in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’14. New York, NY, USA: ACM, 2014, pp. 204–217. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594390

[2] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, “Mining sandboxes,” Saarland University, Tech. Rep., 2015.

[3] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, “AppGuard–fine-grained policy enforcement for untrusted Android applications,” in *Data Privacy Management and Autonomous Spontaneous Security*, ser. Lecture Notes in Computer Science, J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, Eds. Springer Berlin Heidelberg, 2014, pp. 213–231. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54568-9_14

[4] P. von Styp-Rekowsky, S. Gerling, M. Backes, and C. Hammer, “Idea: Callee-site rewriting of sealed system libraries,” in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science, J. Jrjens, B. Livshits, and R. Scandariato, Eds. Springer Berlin Heidelberg, 2013, vol. 7781, pp. 33–41. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36563-8_3

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “PScout: Analyzing the Android permission specification,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222