Published in final edited form as:
Computer (Long Beach Calif). 2016 October ; 49(10): . doi:10.1109/mc.2016.314.

Combinatorial Methods in Security Testing

Dimitris E. Simos[1], **Rick Kuhn**[2], **Artemios G. Voyiatzis**[1], **Raghu Kacker**[2]

[1]SBA Research, Vienna, Austria

[2]NIST, Information Technology Laboratory, Gaithersburg, MD, USA

Introduction

Many software security vulnerabilities result from the exploitation of ordinary coding flaws, rather than design or configuration errors. One study found that 64% of vulnerabilities are the result of such common bugs as missing or incorrect parameter checking, leaving the application open to common vulnerabilities including buffer overflows or SQL injection [1]. While this statistic may be discouraging, it also means that better functionality testing can have the additional benefit of significantly improved security.

Testing that can reveal complex faults that occur only under rare conditions may be especially effective. Empirical data show that most failures are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six, a relationship known as the *interaction rule* [2]. An example of a single-value fault might be a buffer overflow that occurs when the length of an input string exceeds a particular limit. Only a single condition must be true to trigger the fault: *input length* > *buffer size*. A 2-way fault is more complex, because two particular input values are needed to trigger the fault. One example is a search/replace function that only fails if both the search string *and* the replacement string are single characters. If one of the strings is longer than one character, the code does not fail, thus we refer to this as a 2-way fault. More generally, a t-way fault involves t such conditions.

Figure 1 shows the cumulative percentage of faults at $t = 1$ to 6 for various applications studied by NIST and others [3][4][5][6]. As shown in Figure 1, the fault detection rate increases rapidly with interaction strength, up to t=4, reaching 100% detection with 4 to 6-way interactions. Thus, the impossibility of exhaustive testing of all possible inputs is not a barrier to high assurance testing. That is, even though we cannot test all possible combinations of input values, failures involving more than six variables are extremely unlikely because they have not been seen in practice, so testing all possible combinations provides very little benefit beyond testing 4 to 6-way combinations.

The effectiveness of any software testing technique depends on whether test settings corresponding to the actual faults are included in the test sets. When test sets do not include settings corresponding to actual faults, the faults will not be detected. Conversely, we can be confident that the software works correctly for t-way combinations contained in passing

tests. For security evaluations, it is not enough that failures are unlikely to occur in ordinary usage, because attackers seek out even complex flaws. Testing only to verify requirements coverage is insufficient for security, or even for assurance of critical functionality.

Matrices known as *covering arrays* can be computed to cover all *t*-way combinations of variable values, up to a specified level of *t* (typically $t \leq 6$), making it possible to efficiently test all such *t*-way interactions [7]. But any test set, whether constructed as a covering array or not, contains a large number of combinations. We can measure this *combinatorial coverage*, i.e., the coverage of *t*-way combinations in a test set, for a better understanding of test set quality. These measurements contribute quantitative input for risk analysis activities, towards providing answers to questions such as: how many different scenarios have been checked? Are the untested scenarios important? How significant is the risk if we do not increase the test coverage? Do the market share or other external forces (e.g., conformance with standards) justify increasing the test coverage?

Experience report – case studies for software security

SBA Research (https://www.sba-research.org/) and NIST (http://www.nist.gov/) have developed a research program introducing combinatorial testing (CT) based approaches to software security testing. The program aims to bridge the gap between combinatorial testing methods and security testing and, in the process, establish a new research field: *combinatorial security testing*. We describe in the following our case studies and experiences so far.

Parsing untrusted content on the web

The W3C online tidy service[1] is designed to detect and correct HTML code. It accepts a URL through a web form, then parses its HTML source code and reports any fixes that should be made. It has been online for many years now, and has been exposed to multiple instances of malformed code.

In coordination with W3C, SBA Research received permission for an external web application penetration test. We performed an input parameter modeling (IPM) of the Document Object Model (DOM) and generated an attack grammar. The resulting IPM was an enhancement of our previous CT-based approaches for web security testing [11], [12], [13]. The NIST ACTS tool was used then to produce test cases that ensured 100% coverage of the two-way and three-way parameter interactions of the DOM, as recent studies demonstrated its effectiveness for web security testing [14]. We tested the online service against the generated test suite using a prototype XSS injection tool and succeeded in discovering a previously-unknown Remote-Cross-Site-Scripting (RXSS) vulnerability of this popular service.

The sophistication of the attack vector produced by the combinatorial testing technique might explain why the vulnerability has gone unnoticed all this time. W3C has promptly fixed the offending code and acknowledged SBA Research's efforts and responsible

[1] http://services.w3.org/tidy/tidy

disclosure[2]. We note that in this case the source code of the W3C online tidy service was not available (black-box testing) but only a web interface to interact with it.

Web application security

A second example use of automated web penetration testing was demonstrated with the Koha integrated library management system[3]. Koha is used by various organizations, including the UNESCO, the Spanish Ministry of Culture, and the Vienna Cultural Museum. In this case, the source code is available under an open source license. SBA Research developed an IPM towards testing the API of this web-based application. The IPM modelled the parameters passed back and forth encoded as URL parameters. We differentiated two groups of tests, one using a normal (non-privileged user) account of the system and one using an administrator (privileged user) account.

We successfully modelled 43 URLs accepting between 5 and 15 parameters each of various values. We used the NIST ACTS tool to produce test suites that fully covered all possible 2-, 3-, 4- and 5-way parameter combinations. We carried out the testing experiments through the XSS injection tool we used to also test the W3C online tidy service.

We discovered more than 50 cases of XSS vulnerabilities[4] and we reported these to the developers[5 6]. Two related CVEs (CVE-2015–4630, CVE-2015–4631) are now assigned and await final approval by the MITRE team.

System call testing

The kernel of an operating system is its central authority to enforce security features. There exists an extremely large user base (e.g., in 2013 more than 1.5 million Android devices were activated per day[7]). In the case of Linux kernel, some manual testing approaches exist, e.g., the Trinity fuzzer[8] and the Linux test project[9]. Related work in this domain also includes fuzzing techniques for system call testing [15].

We generated IPMs for the Linux system call API by introducing novel combinatorial modelling methodologies [16]. Furthermore, we developed a highly-configurable combinatorial kernel testing framework, namely ERIS, which encompasses automated test execution and logging capabilities. The testing framework allows any test generator to be plugged in for generating automated tests for the Linux system calls. In this regard, we have used the NIST ACTS tool to produce test suites covering a variety of t-way combinations depending on the number of system call arguments. Our testing experiments revealed various erroneous cases that were flagged for further analysis.

2 https://www.w3.org/blog/2014/12/rxss-security-audit-results/
3 https://koha-community.org/
4 https://www.exploit-db.com/exploits/37389/
5 https://koha-community.org/security-release-koha-3-20-1/
6 https://koha-community.org/security-release-koha-3-16-12/
7 http://www.androidcentral.com/android-reaches-900-million-activations
8 http://codemonkey.org.uk/projects/trinity/
9 https://github.com/linux-test-project/ltp

Hardware Trojan detection

Contemporary hardware design shares many similarities with software development practice. The insertion of malicious functionality in hardware (also known as “hardware Trojan horse”) is a realistic threat. The assumptions are that the Trojan activation is controlled using a (short) input pattern out of billions possibilities and the effect of the Trojan’s payload can be observed in the output of the circuit. The attack can be as subtle as introducing a faulty operation on a cryptographic core and deriving the cryptographic key afterwards [8].

Established functional testing techniques from the hardware domain do not cope well with Trojans due to the enormous space of possible input signals used as activation patterns. Modelling the attack as a functional black-box testing problem, a combinatorial testing approach reduced the size of the test suites (and, as a consequence, the testing time) by three orders of magnitude compared to alternative ones and, by construction, guaranteed multiple activations of the Trojan (if existent in first place). This line of research resulted also in new, optimized CAs with interaction strengths beyond six (6) and introduced combinatorial testing as an efficient means not only for software but for hardware security testing as well [9], [10].

Protocol interaction testing

Software implementations of the Transport Layer Security (TLS) protocol specification are a critical component for the security of Internet communications and beyond. Software bugs and attacks still surface in implementations of the TLS protocol. This can be attributed to the complexity of the protocol and its large number of interactions. System designers and integrators are faced with a challenging task: they must ensure that the TLS implementation used in their system can handle correctly all cipher suites (the named combination of various cryptographic algorithms negotiated between a client and a server in TLS), and, at the same time, also conform to a desired level of a security.

We presented a coverage measurement for available TLS cipher suites recommendations [17]. The cipher suites were measured and analyzed using a combinatorial approach and the NIST CCM tool, after deriving the appropriate IPMs. None of the proposed recommendations covered all 2-way combinations of algorithms appearing in a cipher suite; this may be due to incompatibilities or security considerations. Implications for aspects of test quality were also suggested. For example, increasing the number of potential interactions between configuration settings may also increase the risk of bugs or vulnerabilities arising from feature interactions among two or more components. Thus, measuring the level of 2-way, 3-way, and higher strength interactions may be informative for testing.

Conclusions

We have introduced CT-based approaches for security testing and presented our case studies and experiences so far. The success of the presented research program motivates further intensive research on the field of combinatorial security testing. In particular, security testing

for the Internet of Things (IoT) is an area where these approaches may prove particularly useful. IoT systems send and receive data from a large (often continually changing) set of interacting devices and the number of potential communicating pairs increases with the square of the number of devices. Combinatorial methods are ideally suited for the IoT environment, where testing can involve a very large number of nodes and combinations.

References

[1]. Heffley J, & Meunier P (2004, January). Can source code auditing software identify common vulnerabilities and be used to evaluate software security?. In System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on (pp. 10-pp). IEEE.

[2]. Kuhn D. Richard, Kacker Raghu N., and Lei Yu. Introduction to combinatorial testing. CRC press, 2013.

[3]. Kuhn DR, Wallace DR and Gallo AM Jr, 2004. Software fault interactions and implications for software testing. IEEE Trans Soft Eng,30(6), pp. 418–421.

[4]. Bell KZ Optimizing Effectiveness and Efficiency of Software Testing, PhD thesis, North Carolina State University, 2006.

[5]. Cotroneo D, Pietrantuono R, Russo S, & Trivedi K (2016). How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. J.Systems and Software, 113, 27–43.

[6]. Ratliff Z, Kuhn R, Kacker R, Lei Y, Trivedi K, The Relationship Between Software Bug Type and Number of Factors Involved in Failures, submitted to Intl Wkshp Combinatorial Testing, 2016.

[7]. Lei Y, Kacker R, Kuhn DR, Okun V and Lawrence J, IPOG: a General Strategy for T-way Software Testing, 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), Tucson, Arizona, March 26–29 2007, pp. 549–556.

[8]. Bhasin S, Danger J-L, Guilley S, Ngo XT, and Sauvage L. Hardware Trojan horses in cryptographic IP cores. IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2013), pages 15–29, IEEE, 2013.

[9]. Kitsos P, Simos DE, Torres-Jimenez J, and Voyiatzis AG. Exciting FPGA Cryptographic Trojans using Combinatorial Testing. In 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015). November 2–5, 2015.

[10]. Voyiatzis AG, Stefanidis KG, and Kitsos P. Efficient Triggering of Trojan Hardware Logic. 19th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2016).

[11]. Bozic J, Simos DE, and Wotawa F. 2014. Attack pattern-based combinatorial testing. In Proceedings of the 9th International Workshop on Automation of Software Test (AST 2014). ACM, New York, NY, USA, 1–7.

[12]. Bozic J, Garn B, Kapsalis I, Simos D, Winkler S, and Wotawa F, "Attack Pattern-Based Combinatorial Testing with Constraints for Web Security Testing," Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on, Vancouver, BC, 2015, pp. 207–212.

[13]. Garn B, Kapsalis I, Simos DE, and Winkler S. 2014. On the applicability of combinatorial testing to web application security testing: a case study. In Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing (JAMAICA 2014). ACM, New York, NY, USA, 16–21.

[14]. Bozic J, Garn B, Simos DE and Wotawa F, "Evaluation of the IPO-Family algorithms for test case generation in web security testing," Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, Graz, Austria, 2015, pp. 1–10.

[15]. Gauthier A, Mazin C, Iguchi-Cartigny J, and Lanet J-L. 2011. Enhancing Fuzzing Technique for OKL4 Syscalls Testing. In Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security (ARES '11). IEEE Computer Society, Washington, DC, USA, 728–733.

[16]. Garn B and Simos DE, “Eris: A Tool for Combinatorial Testing of the Linux System Call Interface,” Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on, Cleveland, OH, 2014, pp. 58–67.

[17]. Simos DE, Kleine K, Voyiatzis AG, Kuhn R, and Kacker R, “TLS Cipher Suites Recommendations: A Combinatorial Coverage Measurement Approach,” Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on, Vienna, Austria, 2016.

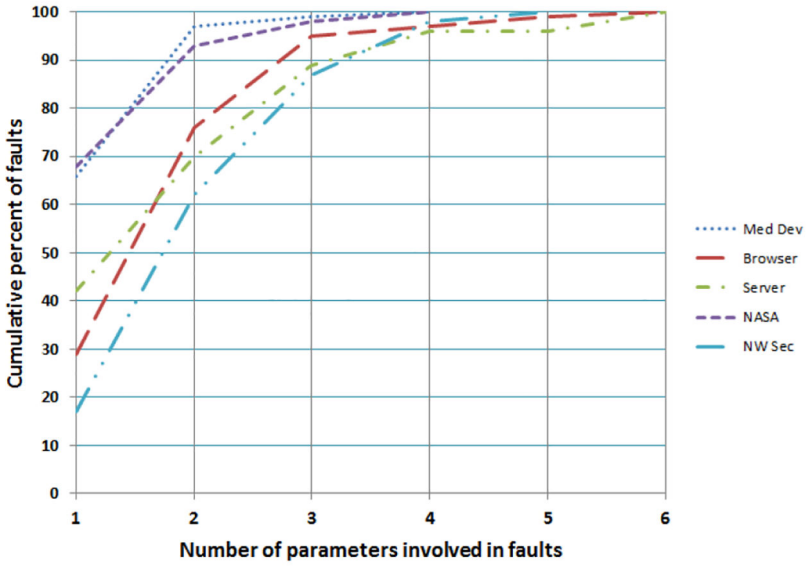

Figure 1.
Cumulative fault distribution