Chapter 7
Security Testing IoT Systems

7.1 Introduction

Systems need to be evaluated for conformance to specifications and requirements, including security, and IoT systems are no exception. Verification and validation techniques are one option to ensure that systems are built according to specifications and requirements, but their use is limited for two main reasons: (i) the complexity of these processes is growing exponentially with the size of the checked system, and (ii) the current business models that include long supply chains with different providers and developers of system components do not enable a unified description of designs and implementations that can be checked as a whole. In the case of IoT systems specifically, the size of most systems is not prohibitive for formal verification and validation methods; however, the lack of a complete implementation with the same tools and models leads to fragmented application of verification techniques to components. Testing constitutes an important and necessary phase in system development, which complements all other approaches and enables the evaluation of integrated systems. Thus, testing is an integral part of the systems development cycle with the purpose to evaluate system correctness, performance, and security at least. Importantly, testing is a method used by customers and certification authorities to evaluate conformance of systems to standards and to provide certifications at the device, system, and product level.

The wide deployment of consumer electronics devices has brought significant attention to testing and its methodologies not only for accepting devices by consumers but also for security, since attackers exploit testing methodologies to identify vulnerabilities and exploit them for their purposes. This is especially important to IoT systems which typically have a cyberphysical component. Identification of vulnerabilities in IoT systems and their exploitation may compromise their safety properties and lead to significant operational problems that result to monetary losses, operation disruption, and even loss of life.

D. Serpanos, M. Wolf, *Internet-of-Things (IoT) Systems*,
https://doi.org/10.1007/978-3-319-69715-4_7

Successful testing of IoT systems is critical considering that many of them have strong requirements that are crucial to their operation, such as meeting real-time constraints, satisfying specific safety properties, and continuing operation even under strained conditions. Furthermore, IoT systems include a communication component, which constitutes a testing challenge because the specifications of communication protocols often have undefined parameters that lead to differing implementations by different vendors; this is the reason why interoperability in communication systems is an important challenge. The criticality of IoT testing, especially for security, becomes more apparent when considering industrial IoT systems, which are extensively used in critical infrastructures nowadays, such as energy networks, water management systems, etc. Successful testing not only confirms the expected operations but takes away from attackers the tools to cause malfunctions and disruptions; in the emerging environment, even crashing an application or an operation may be more catastrophic than hijacking them.

Hardware and software testing are technological areas with significant effort in the market and in academia for decades. A large number of methodologies and tools have been developed, but software testing has been a significantly harder problem than hardware testing because of several differentiating characteristics software has, such as evolution through added features and functionality, fault models and lack of re-use. Considering that most IoT systems are built using off-the-shelf hardware components and computing subsystems, we address software testing for security in this chapter, and, more specifically, we focus on the testing of their communication protocol implementations, since it is the point of entry to systems and a common target of attackers. We present fuzz testing, the most common testing approach for security, which requires no information about the internal structure of the system that is tested. As industrial IoT systems constitute an attractive target for attackers that exploit testing techniques, we use as an example the Modbus protocol and describe fuzz testing techniques for its implementations, which give successful results for existing protocol implementations in the field.

7.2 Fuzz Testing for Security

Vulnerabilities in network systems and applications are identified and disseminated publicly [Nis, Sfo, Str]. The cost of fixing these vulnerabilities can be high, while their exploitation may have quite costly consequences. As a result, there is strong research and development effort to reduce such vulnerabilities.

Static analysis of source code is one approach that does not require program execution but is limited because it does not detect vulnerabilities that are activated by dynamic instruction sequences, during program execution, e.g., dependent on subroutine calls [Che07, Vie00]. Also, these methods present a high false-positive rate leading to significant overhead for the evaluation of the results. Alternatively, dynamic analysis methods intervene in program execution. StackGuard, for example, expands a C compiler and produces executable code that identifies potential execution faults – examining addresses, for example – without changing the functionality of the original programs [Cow98]. TaintCheck applies taint analysis for automatic vulnerability analysis without need for the source code [Cla07, New04]. Dynamic analysis methods enable powerful mechanisms for vulnerability detection at the cost of execution time overhead, because of the additional code that is inserted in the application program. Simulation has also been proposed for vulnerability testing, where a simulation environment is used to inject faults to a program and check its behavior [Du02]. This is a systematic method, but it is limited to input patterns that may cause errors.

Fuzz testing (fuzzing) provides an alternative, reliable approach with successful results and advantages over the previous methods. Fuzzing is a testing method that applies test inputs (vectors) to a system under test (SUT) and observes its outputs, as shown in Fig. 7.1. The goal of the fuzzer is to identify faults in the SUT, e.g., to detect inputs that lead to a system crash. The effectiveness of the fuzzer is based on its ability to identify as many vulnerabilities as possible covering effectively the input value space. If there is inability to identify whether a system or a program has crashed during a test, the effectiveness of the fuzzer cannot be evaluated.

Fuzzing provides several advantages over static and dynamic analysis. First, it can be applied to programs whose source code is not available. Second, it is independent of the internal complexity of the tested software which limits in practice other methods, such as static analysis. Because of this independence, the same fuzzing tool can be used to test similar programs independently of the programming language used for their coding. Finally, the identified faults and errors can be directly associated to the user input and can be evaluated easier.

Fuzz testing has its limitations. The space of input values is vast, and thus, it is impossible to test large systems for all their potential input values within reasonable time frames. A fuzzer that produces random input values can discover faults and vulnerabilities, but, in general, it will not detect easily many important vulnerabilities unless it follows some specific strategic approach. Its effectiveness depends on its ability to identify representative input values, which may originate from attacks or common errors with invalid inputs, and detect vulnerabilities that are useful to attackers.

Fuzz testing can be classified in three (3) categories, depending on the information that is available for the system under test (SUT) [Tak08] [Sut07], as shown in Fig. 7.1:

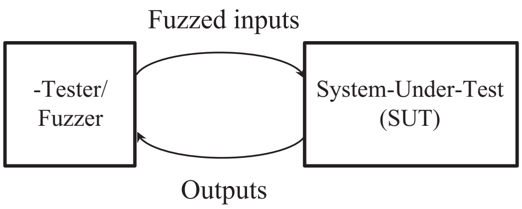


Fig. 7.1 Fuzz testing configuration

- White-box testing: the source code or the specification of the SUT is known.
- Black-box testing: the internal structure of the SUT is unknown –testing is limited to observations of SUT inputs and outputs.
- Grey-box testing: partial information for the SUT internal structure is available, e.g., through reverse engineering or static analysis results.

7.2.1 White-Box Fuzzing

Modern white-box fuzz testing tools exploit the information about the system's internal structure using symbolic execution techniques or taint analysis to identify vulnerabilities. Symbolic execution replaces symbolic values in the source code or the program flow, in order to evaluate code execution paths [Cad13]. These techniques have been explored widely in efforts such as DART [God05], SAGE [God12], EXE [Cad06], and KLEE [Cad08]. Tools like AEG [Avg11] and CRAX [Hua12] combine symbolic execution with concrete execution, employing concolic testing [Sen05] to identify vulnerabilities that lead to control flow hijacking. Such tools have been very successful in fuzz testing of Windows and Linux applications [God12, Cad06]. The techniques have the advantage that they can explore all possible modes of applications, since they use the source code, and identify dead code. However, they cannot identify logic errors in programs and are unable to explore all execution paths in large programs with complex structures. Tools that use taint analysis identify potential attack points in programs by tracing tainted values and then fuzz the input values to these attack points [Sch10]. BuzzFuzz [Gan09] and TaintScope [Wan10] are two representative tools that exploit taint analysis techniques.

7.2.2 Black-Box Fuzzing

Black-box fuzzing techniques do not have any structural information about the system under test. Since testing requires application of inputs to the system and observation of its outputs, one of the most popular targets of black-box fuzzing is the implementation of communication protocols because they provide the first point of entry to systems and they typically implement some standard; so, our description is focused on protocols, although the techniques can be applied to application and system software in general.

There are two main approaches to generate fuzz testing inputs to protocols: (i) data generation and (ii) data mutation [Nal12, Tak08, Sut07]. Data generation techniques create input packets to a protocol implementation either randomly or with a systematic method that takes into account the specifications of the specific protocol. The contents of these packets may be completely random, or they may take into account the structure of the packets, i.e., their fields, and insert either random or special values in the fields, depending on various parameters, such as the system interface or a specific targeted operation. In this case, the specification of the protocol needs to be integrated in the fuzzer. Clearly, the effectiveness of the fuzzing process depends on the successful integration of the protocol specification in the fuzzer, since any problem in that integration may lead to limited or no coverage of a wide range of tests.

Mutation fuzzing creates the test inputs based on legal protocol packets. It takes as input the legal packets and changes (mutates) some of their data, e.g., specific fields, in order to create the test packets that are input to the system. This approach is especially useful in cases where the protocol is complex, because the fuzzer does not construct packets from scratch but uses known legal packets and mutates them. Thus, the fuzzer does not need to include the protocol specification, and the author of the fuzzer does not need to delve into the details of the protocol, thus avoiding the risk of misinterpretations and creation of inappropriate packets.

These two main approaches are coupled with techniques that choose the values that are used in the generated or mutated packets. The most common techniques are:

1. Random: generates of random values without any consideration of packet structure, legal values, etc. The technique is fast, low cost, and quite successful [Mil90, Mil95, Mil06] but limited because it is characterized by low test coverage.
2. Block-based: manages data values in blocks, taking into account the specifications of protocols and creating meaningful blocks of values, in contrast to random values. The technique has been used widely in frameworks and tools, such as Spike [Ait02], SNOOZE [Ban06], Sulley [Ami14], Peach [Pea14], Autodafè [Vua06], and AspFuzz [Kit10], and is especially useful in mutation fuzzing. The success of the technique depends on the successful integration of protocol specs in the fuzzers.
3. Grammar-based: embeds a grammar in the fuzzer, in order to cover part of the specification of legal inputs to the system under test. Fuzzing inputs are created with the consideration of the grammar. PROTOS [PRO] is a representative tool using this technique.
4. Heuristic-based: generates new fuzzing inputs taking into account the effectiveness of the inputs applied in the past. Processing of the outputs obtained from the prior tests can be done with various methods such as with appropriate genetic algorithms [Spa07] or statistical analysis [Zha11].

There exist also approaches that construct protocol descriptions or specifications by observing real protocol traffic. With this information, related tools can make more effective decisions about how to mutate observed packets, in order to increase the effectiveness of mutation fuzzers. General Purpose Fuzzer (GPF) [Vda14] and AutoFuzz [Gor10] are representative tools that employ this approach. Interestingly, in mutation fuzzing there is also the approach of creating test cases based on existing attack traffic [Ant12, Tsa12].

7.3 Fuzzing Industrial Control Network Systems

Fuzz testing for industrial networks has attracted significant interest in the market and in academia, considering the increasing adoption of industrial control systems in critical infrastructures. Many commercial and open source fuzzing tools support industrial protocols. Sulley [Dev07] provides fuzzing modules for ICCP, Modbus, and DNP3 since 2007. ProFuzz [Koc], a fuzzing tool based on Scapy [Bio], supports fuzzing in PROFINET. Achilles test platform [Ach17] supports fuzzing for SCADA protocols, like Modbus/TCP and DNP3.

There is also research work in fuzzing industrial protocols using various techniques. Black-box mutation fuzzing, for example, has been explored for SCADA networks without any knowledge about the networking protocol [Sha11] and using the LZ-Fuzz tool [Bra08] to evaluate its effectiveness. OPC-MFuzzer [Wan13, Qi14] is a mutation fuzzer (based on Peach [Pea14]) for OPC SCADA fuzzing. Based on three different mechanisms to produce fuzzing inputs, the tool identified and confirmed known vulnerabilities that had been included previously in the National Vulnerability Database (NVD) [Nis].

Modbus fuzzing has attracted significant attention as well. BlackPeer [Byr06] produces inputs and checks outputs using a grammar that is included in the tool; although successful, it has limited flexibility as it cannot adjust easily to new tests. Sulley [Dev07], a block-based framework, enables methodical and easy mutation fuzzing through its Modbus module; however, its block-based approach is limited for testing devices that deviate from the standard implementation and are customized by the users. A framework for fuzz testing Modbus for security has also been proposed based on Scapy [Kob07].

7.4 Fuzzing Modbus

7.4.1 The Modbus Protocol

Modbus is an application protocol for industrial control system communication, which has become a standard published by Modbus IDA [Mod, ModS]. Its specification defines the protocol for direct communication over serial links as well as communication over TCP connections. The popular Modbus protocol stacks are shown in Fig. 7.2; it should be noted that, in correspondence with the ISO protocol reference model, Modbus is an application layer protocol defined to interface directly to layer 1 (serial) and layer 2 (HDLC) protocols –stacks (a) and (b) in the figure – or to TCP through an adjusting sublayer that is denoted as Modbus messaging (mapping) on TCP, as shown in stack (c).

The protocol implements client/server (alternatively, master/slave) communication through a request-response model between a control center and field devices, such as a SCADA and PLCs. For example, a SCADA master unit (client) may

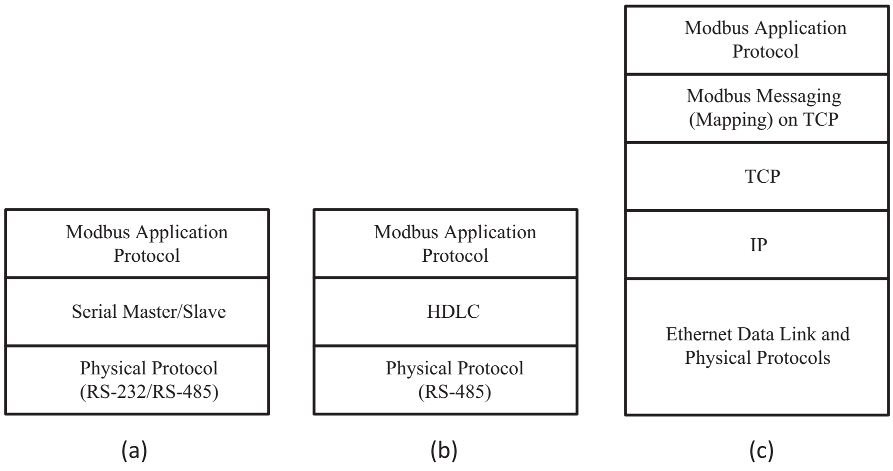
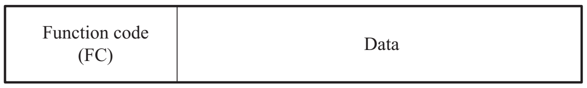

Modbus Application Protocol
Serial Master/Slave
Physical Protocol (RS-232/RS-485)
(a)
Modbus Application Protocol
HDLC
Physical Protocol (RS-485)
(b)
Modbus Application Protocol
Modbus Messaging (Mapping) on TCP
TCP
IP
Ethernet Data Link and Physical Protocols
(c)

Fig. 7.2 Modbus protocol stacks

Fig. 7.3 Modbus application packet



request the reading of a sensor attached to a slave PLC (server), or it may request the writing of a command to an actuator to turn a switch.

Modbus application packets are simple, composed of two fields, a function code (FC) and data, as shown in Fig. 7.3. Requests from servers send the function code that defines the operation to be performed and the related data, e.g., an address or command. A response from a client includes the function code that was executed at the client and the resulting related data. Since an operation may not be successfully executed at the client, the protocol defines that the client will respond with the original function code if the related operation is executed correctly, or it will send an exception code indicating that the operation was not executed.

Modbus has three different classes of function codes: public codes, user-defined codes and reserved ones. Public codes are defined by the standard and include numbering and operation definition. Reserved codes are also public, but they cannot be used freely, since they have been defined and reserved for interoperability purposes with legacy industrial control systems. User-defined codes are available to developers and users to implement specialized function codes at will. Since the function code field is 8 bits, function codes can have 256 values, in the range 0–255. Public codes are in the ranges 1–64, 73–99, and 111–127; these ranges include the reserved codes. User-defined codes may have values in the ranges 65–72 and 100–110. The codes 128–255 are used to indicate errors; each function code has its unique related exception code, which differs from the function code at the most significant bit; with the 8-bit format, all function codes have “0” as their most significant bit and all exception codes have it as “1.”

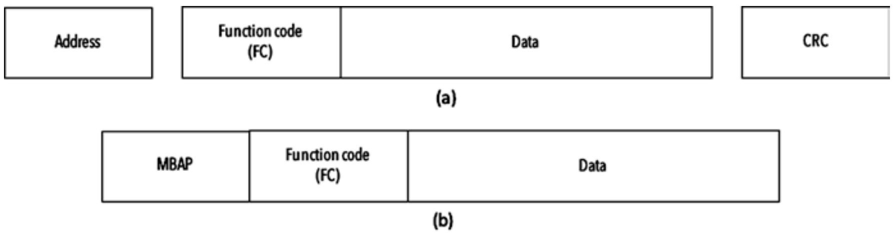


Fig. 7.4 Encapsulated Modbus application packets

Most Modbus function codes perform read and write operations to device data. For this purpose, Modbus considers that devices store data in tables. There are four different table types, based on the data entry size (1 bit or 16 bits) and the access operation allowed (read or read/write). The tables are denoted as (i) discrete input, with 1-bit entries and only read operations allowed; (ii) coils, with 1-bit entries and read/write operations allowed; (iii) input registers, with 16-bit entries and only read operations allowed; and (iv) holding registers, with 16-bit entries and read/write operations allowed. All four types of tables can have up to 64 K entries. Importantly, these tables are actually virtual, meaning that they can be physically separate in the device's memory or they can overlay over the same physical memory cells. Modbus can also access files, which are sequences of records (up to 10,000), and each record has a length measured with 16-bit units.

Modbus application packets (protocol data units, or PDUs) are encapsulated in lower layer protocol packets to be transmitted. When serial connections are used, the application packets are encapsulated by the data link control (DLC) protocol and produce DLC PDUs that are then transmitted by the serial protocol. DLC packets add an address field for the slave next to the function code field and a checksum next to the data field of the application protocol, as Fig. 7.4a shows. In the case of the serial physical layer, there are two formats for the DLC packets, denoted as RDU and ASCII. The main difference between the two is the size of the slave address and the size of the function code field: in RTU format, they are both one byte, while in the ASCII format, each one is 2 bytes long.

Modbus over TCP is performed by extending the Modbus application packet first with an additional header, named MBAP (Modbus Application Protocol) header as shown in Fig. 7.4b, and then encapsulating this extended packet by the TCP/IP protocol stack, which employs Ethernet at the data link control and physical protocol layers, as shown in Fig. 7.2c.

Modbus does not include security mechanisms such as authentication, confidentiality, or integrity. The lack of security renders its implementations vulnerable to a wide range of attacks. The lack of confidentiality enables attackers to extract information from captured packets, while the lack of integrity checks does not allow a receiver of a packet to identify whether the packet has been altered. Replay attacks are possible as well and the lack of non-repudiation mechanisms can lead to inability to analyze and audit systems credibly.

7.4.2 Modbus/TCP Fuzzer

There exist several Modbus fuzzers, as described in Sect. 7.2. In this subsection, we present the approach and results of MTF (Modbus/TCP fuzzer) [Voy15] as a representative example. The choice of MTF is based on its characteristics that show the trends in fuzzing technology today: it is an automated tool, it provides good coverage of input tests, and it does not require physical access to the system under test, operating remotely over the network. These characteristics make MTF an attractive tool for testing security and compliance of Modbus connected devices.

MTF incorporates the specification of Modbus/TCP and supports fuzzing both master and slave devices on the network. As an automated tool for fuzzing, MTF operates in three main phases: (i) reconnaissance, (ii) attack, and (iii) failure detection. In the first phase, MTF identifies the operational characteristics and parameters of the tested system. In the second phase, it applies tests to the system and collects its responses, while in the third phase it evaluates the collected (observed) responses to identify security problems and system failures.

Reconnaissance is an important operation in automated black-box or gray-box fuzzers, because it identifies the operations performed by the system under test and its important parameters. In the case of Modbus, in order to generate meaningful tests, one needs to know the function codes used by the system as well as its memory model, i.e., the four memory types – discrete inputs, coils, input registers, and holding registers – that are specified by the standard. MTF explores the function codes through different methods, in order to accommodate different types of devices that may be fully or partially conformant with the standard. A straightforward method is to ask the device for identification information – the standard specifies function code 43 for this operation – and then, based on this, to find information off-line about the supported function codes, e.g., from a manual. Alternatively, it sends legitimate requests and examines the responses, which indicate whether the requests have been executed or not (as described in the standard specification), or it monitors traffic from the device and extracts functional information from that.

In regard to the memory model of the tested system, MTF effectively identifies the boundary memory addresses for each type of memory. This is done either actively, sending packets with the appropriate function codes probing specific address values, or passively, observing traffic which eventually indicates memory bounds, although these bounds may be approximate.

Taking into account the list of function codes and the memory mapping for the four memory types, the fuzzer can construct legitimate packets and fuzz them in order to test the system. Since the supported function codes are known, MTF constructs a set of packet sequences for each supported function code, where each sequence implements a potential attack to the system; such attacks include packet removal, packet injection, and packet field manipulation.

Packet field manipulation is performed with field values that are boundary, random, or illegal.

When tests are applied, the response, or its absence, is recorded. The tool records the sequence of all tests and related responses and produces a list of errors which are invalid responses (out of specification), valid but with incorrect parameters (values, size, etc.), and delayed or incomplete (no response). Further processing of the records, including both the valid request/response pairs and the errors, leads to detection of security and dependability problems, i.e., malicious or accidental failures.

The MTF approach is representative of the trends in fuzzing industrial protocols. It provides a complete approach to fuzzing, starting with reconnaissance, continuing with meaningful tests and, finally, analyzing the results for security and reliability failures. Its practicality has been demonstrated through the prototype implementation described in the original work [Voy15], which has been used to evaluate several commercial and open source Modbus subsystems and for several attacks. The attacks include packet dropping, packet injection, illegal field values, altered function codes, and even flooding, leading to denial of service attacks. Importantly, many of these attacks have been successful against commercial Modbus implementations, as the reported original results demonstrate. Interestingly, MTF succeeds in attacking these implementations much more efficiently than alternative tools, i.e., with a significantly smaller number of packets. Overall, the results demonstrate that the approach of generation fuzzing is an effective and efficient fuzzing method.

References

[Ach17] Wurldtech- GE Digital, Achilles Test Platform, 2017. https://www.ge.com/digital/sites/default/files/achilles_test_platform.pdf

[Ait02] Aitel, D. (2002). An introduction to SPIKE, the Fuzzer Creation Kit. Presented at The BlackHat USA Conference. www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt

[Ami14] Amini, P. (2014). Sulley: Pure Python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley

[Ant12] Antunes, J., & Neves, N. (2012). Recycling test cases to detect security vulnerabilities. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering*, Dallas, Texas, November 27–30, 2012, pp. 231–240.

[Avg11] Avgerinos, T., Cha, S. K., Hao, B. L. T., & Brumley, D. (2011). AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*, San Diego, California, February 6–9, 2011.

[Ban06] Banks, G., et al. (2006). SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the 9th Information Security Conference (ISC '06)*, pp. 343–358.

[Bio] Biondi, P. Scapy, python interactive packet manipulation framework. http:/www.secdev.org/projects/scapy/

[Bra08] Bratus, S., Hansen, A., & Shubina, A.(2008). LZFuzz: A fast compression-based Fuzzer for poorly documented protocols. Technical Report TR2008–634, Dept. of Computer Science, Dartmouth College, New Hampshire.

[Byr06] Byres, E. J., Hoffman, D., & Kube, N. (2006). On shaky ground – A study of security vulnerabilities in control protocols. In *Proceedings of the 5th International Topical Meeting* *on Nuclear Plant Instrumentation Controls, and Human Machine Interface Technology*, American Nuclear Society, Albuquerque, November 12–16, 2006.

[Cad06] Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., & Engler, D. (2008). EXE: Automatically generating inputs of Death. In: Proceedings of CCS'06, Oct–Nov 2006 (extended version appeared in ACM TIS-SEC 12:2, 2008).

[Cad08] Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI'08*, December 2008.

[Cad13] Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Communications of the ACM, 56*(2), 82–90.

[Cla07] Clause, J., Li, W., & Orso, A. (2007). Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, London, UK, July 9–12, 2007, pp. 196–206.

[Che07] Chess, B., & West, J. (2007). *Secure programming with static analysis*. USA: Pearson Education.

[Cow98] Cowan, C., et al. (1998). StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Usenix Security Symposium*, San Antonio, Texas, January 26–29, 1998.

[Dev07] Devarajan, G. (2007). Unraveling SCADA protocols: Using Sulley Fuzzer. Presented at the DefCon'15 Hacking Conference, 2007.

[Du02] Du, W., & Mathur, A. P. (2002). Testing for software vulnerability using environment perturbation. *Quality and Reliability Engineering International, 18*(3), 261–272.

[Gan09] Ganesh, V., Leek, T., & Rinard, M. (2009). Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 16–24, 2009, pp. 474–484.

[God05] Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, Chicago, USA, June 12–15, 2005, pp. 213–223.

[God12] Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: Whitebox fuzzing for security testing. *ACM Queue, 10*(1).

[Gor10] Gorbunov, S., & Rosenbloom, A. (2010). Autofuzz: Automated network protocol fuzzing framework. *IJCSNS, 10*(8), 239–245.

[Hua12] Huang, S. K., Huang, M. H., Huang, P. Y., Lai, C. W., Lu, H. L., Leong, W. M. (2012). CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. *IEEE 6th International Conference on Software Security and Reliability*, June 20–22, 2012, pp. 78–87.

[Kit10] Kitagawa, T., Hanaoka, M., & Kono, K. (2010). AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In *Proceedings of the IEEE Cymposium on Computers and Communications*, Italy, 2010, pp. 202–208.

[Kob07] Kobayashi, T. H., Batista, A. B., Brito, A. M., & Motta Pires, P. S. (2007). Using a packet manipulation tool for security analysis of industrial network protocols. In *Proceedings of 2007 IEEE Conference on Emerging Technologies and Factory Automation*, Patras, 2007, pp. 744–747.

[Koc] Koch, R. Profuzz. https://github.com/HSASec/ProFuzz

[Mil90] Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM, 33*(12), 32–44.

[Mil95] Miller, B. P., et al. (1995). Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report TR-1268, Department of Computer Sciences, University of Wisconsin-Madison.

[Mil06] Miller, B. P., Cooksey, G., & Moore, F. (2006). An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st International Workshop on Random testing*. Portland, Maine, July 20, 2006, pp. 46–54.

[Mod] ModBus Organization. ModBus Application Protocol Specification http://www.modbus.org/docs/ModbusApplication/ProtocolV11b.pdf

[ModS] Modbus Serial Line Protocol and Implementation Guide V1.02 (Modbus_over_serial_line_V1_02.pdf).

[Nal12] McNally, R., Yiu, K., Grove, D., & Gerhardy, D. Fuzzing: The State of the Art. Technical Note DSTO-TN-1043, Defence Science and Technology Organization, Australia, 02–2012.

[New04] Newsome, J., & Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Technical report CMU-CS-04-140, 2004 (revised 2005).

[Nis] http://nvd.nist.gov

[Pea14] Peach Fuzzing Platform, http://www.peach.tech/products/peach-fuzzer/, 2017.

[PRO] PROTOS-Security Testing of Protocol Implementations. http//www.ee.oulu.fi/roles/ouspg/Protos/

[Qi14] Qi, X., Yong, P., Dai, Z., Yi, S., & Wang, T. (2014). OPC-MFuzzer: A novel multi-layers vulnerability detection tool for OPC protocol based on fuzzing technology. *International Journal of Computer and Communication Engineering, 3*(4), 300–305.

[Sch10] Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *2010 IEEE Symposium on Security and Privacy*.

[Sen05] Sen, K., Marinov, D., & Agha, G. (2005). CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference (held jointly with 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering)*, September 5–9, 2005, pp. 263–272.

[Sfo] http://www.securityfocus.com

[Sha11] Shapiro, R., Bratus, S., Rogers, E., & Smith, S. (2011). Identifying vulnerabilities in SCADA systems via fuzz-testing. *Critical Infrastructure Protection V, IFIP AICT, 367*, 57–72.

[Spa07] Sparks, S., Embleton, S., Cunningham, R., & Zou, C. (2007). Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Proceedings of the 23rd Annual IEEE Computer Security Applications Conference (ACSAC 2007)*, pp. 477–486.

[Str] http://www.securitytracker.com

[Sut07] Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute force vulnerability discovery*. Addison-Wesley Professional.

[Tak08] Takanen, A., DeMott, J., & Miller, C. (2008). *Fuzzing for software security testing and quality assurance*.

[Tsa12] Tsankov, P., Torabi Dashti, M., Basin, D. (2012). SECFUZZ: Fuzz-testing security protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST 2012)*, June 2–3, 2012, Zurich, Switzerland.

[Vda14] VDA Labs, "General Purpose Fuzzer." Rockford, Michigan, 2014, www.vdalabs.com/tools/efs gpf.html

[Vie00] Viega, J., et al. (2000). ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of 16th Annual IEEE Conference Computer Security Applications (ACSAC'00)*, New Orleans, Louisiana, 2000, pp. 257–267.

[Voy15] Voyiatzis, A. G., Katsigiannis, K., & Koubias, S. (2015). A Modbus/TCP Fuzzer for testing internetworked industrial systems. In *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2015)*. Luxembourg, September 8–11, 2015, pp. 1–6.

[Vua06] Vuagnoux, M. (2006). Autodafe: An Act of Software Torture. Swiss Federal Institute of Technology (EPFL), Cryptography and Security Laboratory (LASEC). http://autodafe.sourceforge.net

[Wan10] Wang, T., Wei, T., Gu, G., & Zou, W. (2010). TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. *2010 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2010, pp. 497–512.

[Wan13] Wang, T., et al. (2013). Design and implementation of fuzzing technology for OPC protocol. In *Proceedings of 9th International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Beijing, China, 2013, pp. 424–428.

[Zha11] Zhao, J., Wen, Y., & Zhao, G. (2011). H-fuzzing: A new heuristic method for fuzzing data generation. In *Proceedings of Network and Parallel Computing*, LNCS, Vol. 6985, Springer, 2011, pp. 32–43.