A Modbus/TCP Fuzzer for Testing Internetworked Industrial Systems

Artemios G. Voyiatzis*†, Konstantinos Katsigiannis‡, Stavros Koubias‡†
* SBA Research, Vienna, Austria
avoyiatzis@sba-research.org
† Industrial Systems Institute/RC 'Athena', Patras Science Part building, GR-26504, Platani Patras, Greece
bogart@isi.gr
‡ Department of Electrical and Computer Engineering, University of Patras, GR-26504, Patras, Greece
kkatsigiannis@upatras.gr, koubias@ece.upatras.gr

***Abstract*—Modbus/TCP is a network protocol for industrial communications encapsulated in TCP/IP network packets. There is an increasing need to test existing Modbus protocol implementations for security vulnerabilities, as devices become accessible even from the Internet. Fuzz testing can be used to discover implementation bugs in a fast and economical way.**

We present the design and implementation of MTF, a Modbus/TCP Fuzzer. The MTF incorporates a reconnaissance phase in the testing procedure so as to assist mapping the capabilities of the tested device and to adjust the attack vectors towards a more guided and informed testing rather than plain random testing. The MTF was used to test eight implementations of the Modbus protocol and revealed bugs and vulnerabilities that crash the execution, effectively resulting in denial of service attacks using only a few network packets.

I. Introduction

An essential activity in software, hardware, and system development is testing. The products are tested so as to ensure correct operation or quality in general. There are many forms of software testing. *Static program analysis* requires access to the source or object code of a program and does not actually executes it. *Code review* is a form of static analysis performed by humans.

Penetration testing is an approach that first identifies the unknown system under test (SUT) and then applies known or elaborated attack vectors to test the system's resilience to simulated attacks. Penetration testing requires access to a functional system but not to its source code. The expertise and knowledge needed for penetration testing is not always available within an organization and thus, it may be necessary to expose its systems to third parties.

Fuzz testing or "fuzzing" is a method for testing under time and budget constraints. Fuzzing is "*a method for discovering software faults by providing unexpected input and monitoring for exceptions*" [1]. A recent fuzzing state of the art is provided in [2]. Fuzzing is a highly-automated testing technique that requires no access to the source code or the internals of the system. It can be applied as a first step towards discovering system's identity. Fuzzing uses random inputs, may run for long times, and may be able to test only a few of the possible SUT states. On the other hand, it is rather inexpensive to apply in practice, compared to the available alternative methods, and it can provide initial indications for further examination.

Fuzzers, i.e., software programs for fuzz testing, date back to 1988 [3]. Fuzzers can be applied in many different scenarios, such as program input parameters (environment variables and command line arguments), packets (network protocols), file formats (read by programs as configuration or as input for processing), application content (web applications, browsers), and memory contents (in-memory fuzzing). A disadvantage of implementing fuzzers is that a client must be developed for each SUT, which is not a portable, re-usable solution. Fuzzing frameworks can overcome this limitation but once they become too generic, the implementation time and complexity raises again.

Embedded systems are commonly found in industrial settings. Such systems cannot run or be attached to software monitoring applications, such as a memory consumption inspectors, due to physical limitations (field installation) and lack of appropriate tools. Thus, fuzzing frameworks that monitor system execution must use other means of execution and must run in separate, independent systems.

Industrial (control) systems comprise many embedded systems. As industrial systems get interconnected and internetworked, they are becoming a very attractive target for attackers. Such systems have been originally developed with a mindset of operating in a disconnected environment, where network connectivity was not a threat. However, these assumptions do not hold anymore.

Fuzzing the implementations of industrial network protocols, such as the Modbus protocol, is an important first step towards getting insights on the resilience of the systems against new threats arising from the Internet. Such systems are an ideal fit for fuzz testing: the source code is not available, the systems were developed years ago, the protocol specifications are open, different vendors and a variety of protocol implementations already exist, the system specification may not be available, and interaction with such systems through network exchanges is the norm.

In this paper, we describe the architecture and the imple-

978-1-4673-7929-8/15/$31.00 © 2015 IEEE

mentation of a fuzzer for testing implementations of the Modbus network protocol when running over TCP/IP. The fuzzer operates in phases so as to minimize the produced network traffic and adapts the fuzz attempts based on the knowledge it collects about the SUT. We tested the fuzzer against eight Modbus implementations in software. The fuzzer succeeded in revealing various deviations from protocol specification, bugs, and software crashes that resulted in a denial of service attack from the side of the SUT. This is an important factor to consider in an industrial setting, as system availability is of utmost importance.

The rest of this paper is organized as follows. Section II discusses network protocol fuzzing, the Modbus protocol, and its security threats at a protocol level. Section III describes the design and implementation of the Modbus/TCP protocol fuzzer we developed, while Section IV presents the experiments held and the performance of our fuzzer. Section V provides the conclusions drawn and the future directions of the reported work.

II. Literature review

A. Network protocol fuzzers

Fuzz testing can be utilized to automatically check an implementation of a network protocol against erroneous (out of specifications) packets received from the network. Compared to exhaustive, methodological search, fuzz testing can provide a faster, albeit randomized, searching for detecting possible points of failure.

The knowledge initially collected by a fuzzer can be later utilized, depending on the application scenario, so as to guide the vulnerability testing towards more "interesting" paths that are more probable to be vulnerable and need fixing. Furthermore, fuzz testing can inject network traffic in a way that cannot be described in rules and thus, its operation may remain hidden from security monitoring appliances.

Network protocol fuzzers can be classified into many categories. A first criterion is the availability of source code that can be analyzed and guide searching. Fuzzing can benefit security testing of heavyweight and complex network protocols, as it can identify specific areas of interest for more focused testing [4]. Symbolic execution is a popular approach for applying the so-called "whitebox" fuzzing [5], [6], [7], [8].

Knowledge of the packet format (blocks) of a network protocol can be used to build mutations for testing [9], [10], [11], [12]. The generation of test packets can be totally random-based or be based on mutations of known (previously captured) network packets. The General Purpose Fuzzer (GPF) is an example of the latter [13].

At a higher level of abstraction, a fuzzer can use the knowledge of the protocol specification (if this is available) and build faulty inputs for injecting traffic in the network [14]. Many network protocols are based in the concept of the "state", engaging in lengthy data exchanges. Stateful fuzzers are used to test such implementations at a deeper level of interaction, including sending valid but out-of-order requests and responses [15], [16], [17].

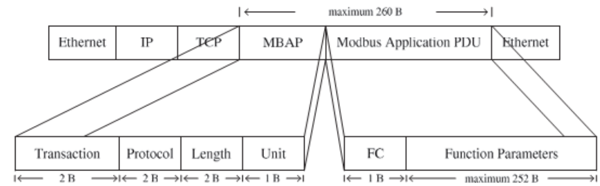

Fig. 1. Modbus frame encapsulation in TCP segments

B. The Modbus protocol

Modbus is a serial communication protocol for industrial control systems published by Modicon (now Schneider Electric) in 1979. It has become the *de facto* standard for connecting industrial electronic devices. Since 2004, the development and update of the Modbus protocol is managed by the Modbus Organization (http://www.modbus.org/). Modbus is a simple and robust protocol, with two roles (master and slave) and stateless communication of request/response frame pairs. Modbus frames can be carried over serial links or TCP/IP.

A common application scenario is a Supervisory Control and Data Acquisition (SCADA) system collecting information from Remote Terminal Units (RTU), such as Programmable Logic Controllers (PLC). In this setting, the SCADA acts as a Modbus *master*, issuing *requests* to its *slave* PLCs that provide the *responses*.

The Modbus slave device is modelled as a set of four memories, namely *coils*, *discrete inputs*, *holding registers*, and *input registers*. The control loops and the reporting can be modeled as a series of reads and writes of these memories, either from the physical processes themselves or through remote commands issued by the Modbus master.

In the case of Modbus over TCP/IP, the Modbus slave (e.g., a PLC device) acts as a *TCP server* waiting for incoming connections at the IANA-assigned port TCP/502, while the Modbus master (e.g., a SCADA system) acts as a *TCP client* connecting to it.

The format of a Modbus frame when transmitted over TCP/IP is depicted in Fig. 1. The Modbus/TCP Application Data Unit (ADU) consists of a 7-byte header (Modbus Application Header, MBAP) and a Modbus Protocol Data Unit (PDU) of up to 253 bytes.

The MBAP consists of a 2-byte transaction identifier; a 2-byte protocol identifier (set to 0x0000 for Modbus); a 2-byte length field, indicating the number of the following bytes; and a 1-byte unit identifier (set to 0xFF, equivalent to the slave address in the serial version of Modbus). In Modbus/TCP a master may have multiple pending transactions with slaves and a slave may be communicating with multiple masters.

The Modbus PDU carries requests that are defined in *function codes* (FC), ranging from 0 to 127 (defined in the Modbus standard: 1–64, 73–99, 111–127), while negative responses range from 128 to 255. Each function code can be followed by *function parameters*, passing request parameters to the slave. For example, a *read coil* function code is followed by the address (number) of the specific coil to be read.

C. Problem statement

Modbus is a rather simple protocol and its implementation *should be* straightforward. However, past assumptions of a strict and protected environment in which industrial control systems operate may be inherently embedded in an implementation. Earlier attempts to analyze the Modbus protocol have revealed many areas of concern for operating Modbus in modern, internetworked environments. The concerns range from reconnaissance to process integrity and denial of service attacks [18].

Some earlier works focus on applying fuzz testing in industrial network protocols, either by extending general-purpose tools (e.g., Sulley for ICCP, DNP3, and Modbus [19]) or developing new ones from the scratch (e.g., for PROFINET, IEC61850, DNP3, Modbus, and OPC [20], [21], [22], [23], [24]). These tools are pure fuzzers, randomly generating or mutating Modbus traffic and manipulating the protocol fields. As such, they may inject a flood of packets for achieving a good coverage and take too long time to execute. The general-purpose tools abstract most of attack details and thus, require adaptation to the specifics of the Modbus protocol. Also, they require access to the SUT so as to detect a crash or violation.

In the following, we describe the Modbus/TCP Fuzzer (MTF), a fuzz testing software we developed. The MTF does not need direct access to the SUT but just a network connection to it. Through the network behavior of the SUT, it can derive if the SUT has crashed or is no longer reachable.

III. Fuzzer design and implementation

The Modbus/TCP Fuzzer (MTF) is a fuzzer that can be used to test both the master and the slave sides of the protocol. Armed with the open specification of the Modbus/TCP protocol, we aimed for an *informed* fuzzer that constructs almost-valid protocol packets (frames) with erroneous inputs. The MTF first builds a list of possible cases in the form of Modbus memory map boundaries and supported function codes. The MTF then uses this collected knowledge so as to perform a *guided* fuzzing and reduce the injected frames and the number of the tests. The MTF aims to minimize generated traffic and reduce testing time in order to reduce network noise and remain as stealth as possible.

A. Reconnaissance

A unique characteristic of MTF is that it incorporates a "reconnaissance" phase. During this phases, the MTF collects information about the SUT in order to adapt its attack strategy to match the capabilities of the SUT and inject as little as possible network traffic. This phase is realized with three different means. As a first option, the MTF connects to a target slave system and enumerates supported Modbus function codes (FCs). This can achieved by sending a Modbus request with `FC=43`. Depending on the level of conformance of the device to the Modbus specification, the slave system *must* respond with a device identification banner and with a list of the FCs it supports. As a second option, the MTF sends legitimate ADUs to the device. These ADUs are carefully selected so as not to produce any action to the system (e.g., no FCs that involve memory writes are sent). Finally, as a third option, the MTF can parse a file with captured network traffic in PCAP format (cf. http://en.wikipedia.org/wiki/pcap/) and construct the (partial) list of supported functionality based on the identified protocol exchanges. This file may originate from the target network and could have been generated by other means beforehand.

Another mapping performed during the reconnaissance phase related to dumping the contents of the four Modbus memory types (coils, discrete inputs, holding registers, and input registers). The memory mapping of the device can be performed either passively or actively. In the active mode, the MTF generates read queries for memory locations so as to discover the upper and lower boundaries for each of the four memory types. The MTF assumes that the addresses of each memory type are allocated in a continuous space and thus, an exhaustive address-by-address search is not necessary. This results in faster execution and in reduced network traffic, allowing MTF to remain as stealth as possible. As some devices may drop the connection upon receiving invalid addresses, the MTF keeps track of the scanning progress and reopens the connection, if necessary. The passive mode is activated when MTF parses passively captured traffic. In this case, the Modbus request and response packets are processed so as to identify the approximate memory boundaries for each memory type.

Once the reconnaissance phase concludes, the MTF has an initial map of the system-under-test (SUT) and its supported functionality. The findings of the reconnaissance phase, independently of the option used, are written in comma-separated value files (CSV) that are then parsed during the attack phase that is described in the following. Armed with this knowledge, the MTF customizes the attack phase so as to match the capabilities of the SUT. This approach can possibly lead to deeper execution paths of the software loaded in the SUT, due to deeper protocol parsing, and thus, reveal hard-to-spot implementation bugs.

B. Attack

In the "attack" phase, the MTF generates valid requests or responses and then fuzzes them. In that sense, the MTF is a *generative* fuzzer. The fuzzed Modbus packets are then injected in the network and the reaction of the SUT is recorded so as to evaluate its robustness to malformed inputs. The packet modification can occur in the payload, the PDU, the MBAP, or the PDU and MBAP parts combined. A fifth option is to perform no fuzzing at all and inject valid packets into the network.

At first, MTF parses the output file of the reconnaissance phase and constructs the list of supported FCs. For each supported FC, a configurable number of packets are prepared. At each fuzzing iteration of each FC, MTF chooses one type of attack (e.g., insert random PDU, remove a valid PDU, change protocol ID field, or request address out of bounds) and records the response or the lack of it. During the fuzzing process, it is possible to change the FC to one that is considered

unsupported. This allows to both fuzz test the SUT and to discover functionality that was not uncovered during the reconnaissance phase.

We performed an initial analysis of the Modbus protocol specification and we identified a list of valid protocol interactions and states that can be reached using a misbehaving or malicious Modbus client so as to drive the receiving end at an unexpected state in its execution flow. Our hypothesis is that that processing information for such states will fire a bug in the software implementation, resulting in a crash, a freeze, or a security violation (e.g., returning memory contents that should not be made available to the requester). We then enriched our list with attacks described already in the published literature. Our list includes the following attack vectors:

- A master sends reply frames instead of request frames.
- A master sends request frames containing an error code (i.e., a function code greater than 127).
- A master sends frames with slave address between 248 and 255 (reserved for future use) or a slave responds with such an address.
- A master sends frames with function code zero (undefined).
- A master sends "write coil" requests with value not equal to 0x0000 or 0xFF00.
- A slave responds to master broadcast commands. The Modbus protocol dictates that broadcast frames must go unanswered.
- A slave swaps role in the middle of a transaction and starts sending request frames.
- A slave sends reply frames with different address than the one addressed to.
- Frames are injected with MBAP field not equal to zero.
- Frames are injected with erroneous MBAP length field.
- Frames are injected with MBAP unit identifier field not equal to 0xFF.
- Frames are injected with more than one ADU per TCP segment.
- Connection flooding: A master exhausts the connection pool of the slave device (not a protocol-level attack).
- Transaction flooding: A master or a slave exhausts the MBAP transaction pool of the other device by changing the transaction number so as to keep the transactions pending (denial of service attack).

We do not claim our list to be an exhaustive one - it is possible that many more attack vectors may exist on a given implementation of the Modbus protocol. However, we claim that it is a good starting point of attack vectors that can confuse a legitimate Modbus protocol implementation operating under the assumption that the other end remains strictly compliant to the protocol specification.

C. Failure detection

The MTF records each request and response for further processing. Two log files, `info` and `error`, are created at each execution. The former is a detailed execution trail while the latter contains the evaluation of the SUT's behavior. Recorded information include: responses outside the Modbus specification, delayed responses or freeze of communication (socket timeout, reset, or close; failure in reopening a closed socket; and failure in opening a new socket at all), valid but incorrect responses (e.g., getting back 12 coil readings when asked for only 10 or getting back different memory readings).

The collected information are then evaluated for detecting failures and security problems. The major concern here is the possibility of denial of service attacks. This can be identified in the logs as a TCP socket close, as inability to open a new socket to the SUT, and as socket timeout errors, which indicate that the SUT is not responsive.

D. Implementation

The MTF is implemented in the Python programming language. The `modbus-tk` open source Python implementation of the Modbus protocol (available at http://code.google.com/p/modbus-tk) and the `pymodbus` implementation (available at https://github.com/bashwork/pymodbus) are utilized to generate valid Modbus/TCP packets and handle the low-level connectivity details between the master and the slave. The Scapy framework is used for manipulating Modbus/TCP packets in an informed way [25].

In our experiments (reported in Section IV), the running time of the reconnaissance phase was under one minute for almost all of the cases and the active part of it introduced less than 2,000 Modbus frames.

IV. EXPERIMENTS AND RESULTS

A series of experiments was held in a testbed environment. We tested the robustness of various available implementations of Modbus. More specifically, MTF was used to test the following implementations:

- `MOD_RSSIM` v8.0.1, a Modbus RTU and TCP/IP simulator (cf. http://www.plcsimulator.org/).
- `XmasterSlave` v2.2015.2.11, a simulator for Windows supporting Modbus, DNP3, and IEC 60810-5-101/103/104 (cf. http://xmasterslave.tgscada.com/).
- `pymodbus` v1.2.0, a Modbus implementation in Python (cf. https://github.com/bashwork/pymodbus).
- `LibModbus` v3.0.6, a Modbus library written in C running on Linux, MacOS X, FreeBSD, QNX, and Win32 (cf. http://libmodbus.org/documentation).
- `Mblogic MB Release 25` (MBAsyncServer v2.0.1), a full platform for industrial automation written in the Python programming language that supports Modbus (cf. http://mblogic.sourceforge.net/mbapps/apps.html).
- `Modbus Slave/Modbus Poll` v6.0.2, a Modbus implementation supporting up to 32 slave devices in Win32 (cf. http://www.modbustools.com/index.asp).
- `FieldTalk`, a commercial C++ library for Windows and Linux supporting serial and TCP connectivity (cf. http://www.modbusdriver.com/). Also, `Modpoll` and `Diagslave` simulator for Windows and Linux that are based on FieldTalk.

- `Communication Protocol Test Harness` v3.17 by Triangle Microworks (TMW), a Windows application acting as Modbus master and slave for testing implementations (cf. http://www.trianglemicroworks.com/products/testing-and-configuration-tools/test-harness-pages).

A. Results of MTF

The MTF created a denial-of-service (DoS) attack on `MOD_RSSIM` and `XmasterSlave` by opening new sockets to port TCP/502 and sending fuzzed packets with MBAP length equal to 1. This resulted in repetitive socket freezes, up until no more new connections could be handled anymore. Less than 200 frames and 1 minute of interaction was needed to reach this stage. Similar results where observed for `pymodbus`; the DoS was realized with less than 10 fuzzed frames.

`LibModbus` and `Mblogic MB` handled better the connectivity issues. Yet, their responses were not valid (e.g., wrong transaction identifiers and responses with out of bounds FCs). This may be an indication of memory corruption. `Modbus Slave` reacted with socket resets and out-of-spec responses when receiving fuzzed requests with MBAP length equal to 259, 1462, and 65534. On the other hand, the diagnostics tools based on the `FieldTalk` library exhibited a stable behavior and were not affected by the fuzzer.

In all the aforementioned cases, no specific attack vectors were responsible for generating the socket freeze and consuming the available socket connections up to the point of a denial of service attack. Rather, it was the whole group of fuzzed packets that broke up the protocol processing after some time.

The MTF managed to freeze and crash the `TMW` software during both the reconnaissance and the attack phase. During the former, a request with `FC=0x2B` resulted in no response and in 100% CPU usage of the machine hosting the software, effectively creating a denial-of-service attack. The only option to resume operation was to terminate the process. The same behavior was observed when sending multiple PDUs in one TCP frame or when inserting a random PDU. In some cases, the operating system was able to terminate the TMW process, resulting in a crash, as depicted in Fig. 2. Since we had no access to the source code of the executable, we were not able to further pinpoint the root cause of the freezes and crashes.

B. Fuzzer comparison

We tested two more fuzzers on the same testbed for the sake of comparing the observed performance of our MTF. The first one is the General Purpose Fuzzer (GPF) [13]. GPF supports the following modes: PureFuzz (randomly-sized Modbus frames); MainGPF (buffer overflow attacks by increasing the size of the packet and logic error attacks by injecting frames with logical errors); and PatternFuzz (mutating captured frames). The second one is the commercial product beSTORM v5.3.1 (the trial version we had available allows as time-limited execution of 30 minutes). Table I summarizes the performance of the three fuzzers (MTF, GPF, and beSTORM)

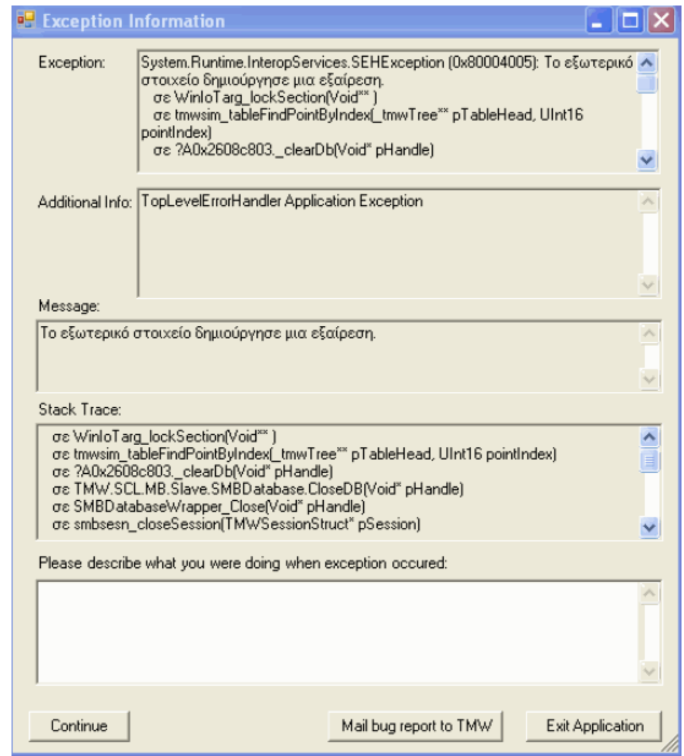

Fig. 2. Screenshot of MTF fuzz result on TMW software

in activating bugs in the eight Modbus implementations based on number of vectors and time needed. We can observe that the MTF requires at least an order of magnitude less frames to be injected in the network in order to manage to freeze the tested Modbus implementations. The only exception to this rule is the case of `FieldTalk`, where the MTF did not result into socket resets, while GPF achieved this after sending about 30,000 frames.

V. Conclusions and Future Work

The Modbus/TCP protocol is used in industrial systems that are no more isolated but rather become more and more interconnected and internetworked. Such systems are an attractive target for attackers. Fuzz testing can quickly and efficiently test the systems for possible bugs and for revealing security weaknesses that may lead to denial-of-service attacks. We designed and implemented MTF, a Modbus/TCP Fuzzer, that incorporates a reconnaissance phase. The MTF enhances the fuzzing approach for industrial systems by performing guided fuzzing. This results in less injected traffic and faster execution. The MTF has been successful in revealing bugs in various existing Modbus/TCP implementations or even software crashes, creating effectively a denial-of-service attack through malformed network packets.

As a future work, we aim to apply MTF against other implementations of the Modbus protocol, with emphasis on commercial devices available in the market. Also, to further enhance the failure detection capabilities of the MTF and to

TABLE I
FUZZER PERFORMANCE COMPARISON (NUMBER OF FRAMES AND TIME UNTIL CONNECTION FREEZE).

Implementation	MTF	GPF	beSTORM
MOD_RSSIM	200 frames; 1 minute	200 frames; 1 minute	200 frames; 1 minute
pymodbus	10 frames	14,000 frames	no freeze
LibModbus	invalid responses	20,000 frames	invalid responses
XmasterSlave	200 frames; 1 minute	1,000 frames; must restart	90,000 frames; 30 minutes limit
Mblogic MB	invalid responses	150,000 frames	80,000 frames
Modbus Slave	600 frames; 9 minutes	9,500 frames; exception code 10	37,000 frames; exception code 10
FieldTalk	no freeze	30,000 frames; crash	crash
TMW	crash	25,000 frames; crash	30 minutes limit; no crash

develop appropriate countermeasures for detecting and isolating fuzzing activities at the network layer, long before they can reach vulnerable devices already installed and operating in the field.

ACKNOWLEDGMENT

This work was partially supported by the GSRT Action "KRIPIS" of Greece with national and EU funds in the context of the research project "ISRTDI" and by the COMET K1 program by the Austrian Research Funding Agency (FFG).

REFERENCES

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[2] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," Defence Science and Technology Organisation, Department of Defence, Australian Government, Edinburgh, South Australia 51111, Australia, Tech. Rep. DSTO-TN-1043, February 2012.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, N. Nystrom, and W. Wang, "Simfuzz: Test case similarity directed deep fuzzing," *Journal of Systems and Software*, vol. 85, no. 1, pp. 102–111, 2012.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[6] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[7] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation." in *NDSS*, vol. 11, 2011, pp. 59–66.

[8] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 78–87.

[9] P. Amini, "Sulley: Pure python fully automated and unattended fuzzing framework." [Online]. Available: https://github.com/OpenRCE/sulley

[10] "Peach fuzzing platform." [Online]. Available: http://peachfuzzer.com/

[11] D. Aitel, "An introduction to spike, the fuzzer creation kit," Presentation at The BlackHat USA Conference 2002. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt

[12] M. Vuagnoux, "Autodafé: An act of software torture," 22nd Chaos Communications Congress, Berlin, Germany, 2005.

[13] V. Labs, "General purpose fuzzer." [Online]. Available: www.vdalabs.com/tools/efsgpf.html

[14] "Protos - security testing of protocol implementations." [Online]. Available: http//www.ee.oulu.fi/research/ouspg/protos/

[15] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *Information Security*. Springer, 2006, pp. 343–358.

[16] T. Kitagawa, M. Hanaoka, and K. Kono, "Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols," in *Computers and Communications (ISCC), 2010 IEEE Symposium on*. IEEE, 2010, pp. 202–208.

[17] P. Tsankov, M. T. Dashti, and D. Basin, "Secfuzz: Fuzz-testing security protocols," in *Automation of Software Test (AST), 2012 7th International Workshop on*. IEEE, 2012, pp. 1–7.

[18] P. Huitsing, R. Chandia, M. Papa, and S. Shenoi, "Attack taxonomies for the modbus protocols," *International Journal of Critical Infrastructure Protection*, vol. 1, pp. 37–44, 2008.

[19] G. Devarajan, "Unraveling scada protocols: Using sulley fuzzer," Presentation at the DefCon 15 Hacking Conference, 2007.

[20] R. Koch, "Profuzz." [Online]. Available: https://github.com/HSASec/ProFuzz

[21] M. Dynamics, "Mu test suite." [Online]. Available: http://www.mudynamics.com/products/mu-test-suite.html

[22] B. Security, "bestorm software security testing tool." [Online]. Available: http://www.beyondsecurity.com/bestorm.html

[23] T. Wang, Q. Xiong, H. Gao, Y. Peng, Z. Dai, and S. Yi, "Design and implementation of fuzzing technology for opc protocol," in *Intelligent Information Hiding and Multimedia Signal Processing, 2013 Ninth International Conference on*. IEEE, 2013, pp. 424–428.

[24] X. Qi, P. Yong, Z. Dai, S. Yi, and T. Wang, "Opc-mfuzzer: A novel multi-layers vulnerability detection tool for opc protocol based on fuzzing technology," *International Journal of Computer and Communication Engineering*, vol. 3, no. 4, July 2014.

[25] T. H. Kobayashi, A. B. Batista, A. Brito, and P. Motta Pires, "Using a packet manipulation tool for security analysis of industrial network protocols," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*. IEEE, 2007, pp. 744–747.