Analysis of Vulnerability of IPsec Protocol Implementation Based on Differential Fuzzing

Kai Tian[1], Fushan Wei[1], Chunxiang Gu[1,2](✉), and Yanan Shi[1]

[1] Information Engineering University, Zhengzhou 450001, China
[2] Henan Key Laboratory of Network Cryptography Technology, Zhengzhou 450001, China

Abstract. Network protocol is an important means to ensure network security, but it has suffered a steady stream of attacks in recent years due to its implementation complexity and difficulty. In this paper, we present our work on using differential fuzzing to detect the behavioral divergences in multiple implementations of IPsec. The key insight behind our fuzzer is to generate various message streams compose of mutate packets and send them to the IPsec implementations to compare their different behaviors. We proposed a protocol testing framework based on differential fuzzing testing, which can be applied to test the differences and potential security issues of multiple implementations. Our case reveals the implementation differences between four protocol implementations. These differential behaviors exposed protocol implementation violations of RFC specifications and possible security vulnerabilities.

Keywords: IPsec · Protocol fuzzing · Differential fuzzing · Software security

1 Introduction

As an important guarantee for network communication, the correct implementation of network security protocols has become a key link to protect user's privacy and information transmission security. However, it's a challenge to implement the network security protocol correctly due to its complexity and diversity. With the rapid development of Internet technology, the scale of network attacks such as trojan, worm, ransomware becomes much larger and the attack means are more various, making the network's communication security face a huge test. Among them, there are many vulnerabilities caused by the implementation of network security protocols, such as the well-known Heartbleed bugs [1], a serious vulnerability in the popular OpenSSL cryptographic software library.

IPsec [2] is widely used to ensure end-to-end secure communication scenarios. It is not a separate protocol, but a complete set of architecture applied to network data security on IP layer, including AH, ESP, IKE, etc. Under reasonable configuration, IPsec protocol is considered to be secure. However, security vulnerabilities often arise not only from the protocol itself, but also from the implementation of the protocol. Among the many software vulnerability discovery techniques, fuzzing has remained highly popular.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2022
Published by Springer Nature Switzerland AG 2022. All Rights Reserved
W. Shi et al. (Eds.): SPNCE 2021, LNICST 423, pp. 282–293, 2022.
https://doi.org/10.1007/978-3-030-96791-8_21

However, it's also difficult to find vulnerabilities in protocol using fuzzing. On the one hand, the server has a large state space, and the whole state space can be traversed only by using the message sequence conforming to the protocol specification. On the other hand, strict semantic format requirements and complex cryptographic algorithms further increase the difficulty and complexity of protocol fuzzing. Existing fuzzing algorithms mostly rely on gray box testing technology guided by code coverage, and usually do not rely too much on the semantic format of the protocol, which is hard to handle complex encryption without modifying the server source code.

We propose our approach, which treats the server as a black box without code instrumentation, generates most effective messages through streams fuzzing and field mutation, and discovers the differences and bugs by observing the feedback of different server implementations.

To summary our contributions:

- We design an IPsec protocol fuzzing frame based on differential fuzzing, which can find the differences between protocol implementations naturally.
- Our test case generation method can automatically generate test cases including certificates without modifying the source code of the protocol implementation, which is more fit into the fuzzing test of the protocol.
- We have implemented our fuzzing tool and applied it to IPsec (IKEv1). Through differential fuzzing, we can find more semantic bugs in the protocol, which is also very important for protocol security analysis.

2 Background

2.1 IPsec

IPsec provides high quality, cryptographically-based security services at the IP layer including access control, connectionless integrity, data origin authentication, protection against replays and confidentiality (encryption) [2], which is widely used in various VPNs.

The protocol for key establishment is known as IKE (Internet Key Exchange), and there are two versions of IKE: IKEv1 [3] and IKEv2 [4]. IKEv1 seems to be much more complex which is split into two distinct phases. In phase 1 an ISAKMP SA is established which is used in phases 2 to set up an IPsec SA. IKEv2 provides the same level security with less interaction.

In this paper, we mainly study the implements of IPsec with IKEv1. In IKEv1, there are four authentication methods in phase 1: Pre-Shared Key based method, signature-based method, and two RSA encryption-based methods.

IKEv1 phase 1 can be divided into main mode and aggressive mode. The main mode includes six messages and aggressive mode includes three messages.

In the main mode, the first two messages are used to negotiate a proposal and provide their own *cookies*: c_I and c_R. The *cooike* value can be used to mark the conversation, which exists in the header of each message. Messages 3 and 4 are mainly used for key establishment according to the selected authentication method. The parameters passed

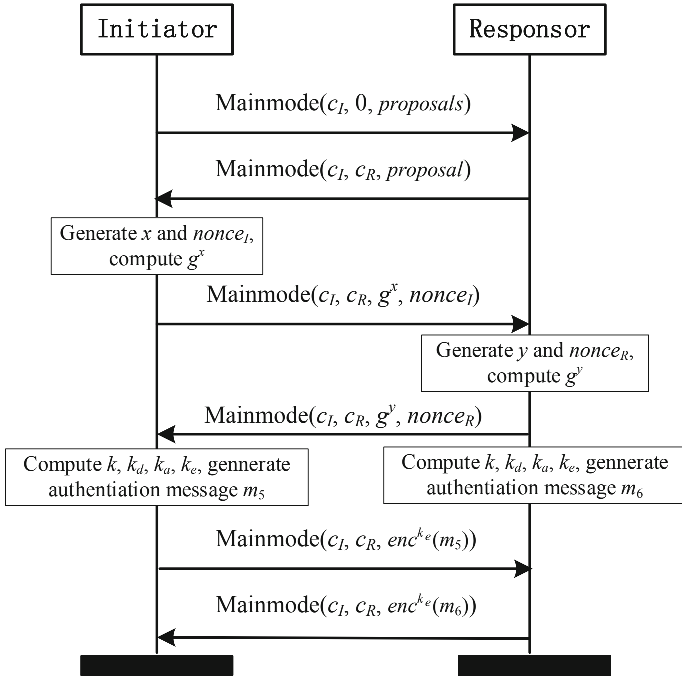

Fig. 1. Generic structure of IKEv1 Phase 1 in main mode.

in this process include DH key exchange parameters and corresponding auxiliary data according to the selected authentication method. Based on these messages and the shared DH secret, the initiator and the responder calculate the four symmetric keys (k, k_d, k_a, k_e) separately. The last two messages are used to confirm each other's keys, which are respectively encrypted with k_e, and the first phase of IKEv1 ends (Fig. 2).

The quick mode is mainly used to negotiate the security parameters used by IPsec SA, which contains three messages, and the quick mode is protected by IKE SA. The initiator will send the conversion attributes of the IPsec SA in the first message, which includes the hash value $hash_1$, the policy proposal SA, and the Nonce. The second message is similar to the first message and the third message contains a hash value, which is used to confirm the receiver's message and prove that the initiator is active. If PFS is desired, a DH Key Exchange can additionally be performed [5].

At this point, a complete connection is established between the inititor and the responsor. Our experiment mainly focuses on the signature-based method and PSK based method. Below we briefly introduce two authentication methods.

PSK-Based Authentication. PSK-based authentication requires both parties to know the same password. The two parties in communication use the shared key, the nonce during the connection, and the shared DH secret to derive the key (refer to the Fig. 1), and the calculation of the key methods is as below:

$$k = prf_{\mathrm{PSK}}(nonce_I, nonce_R)$$

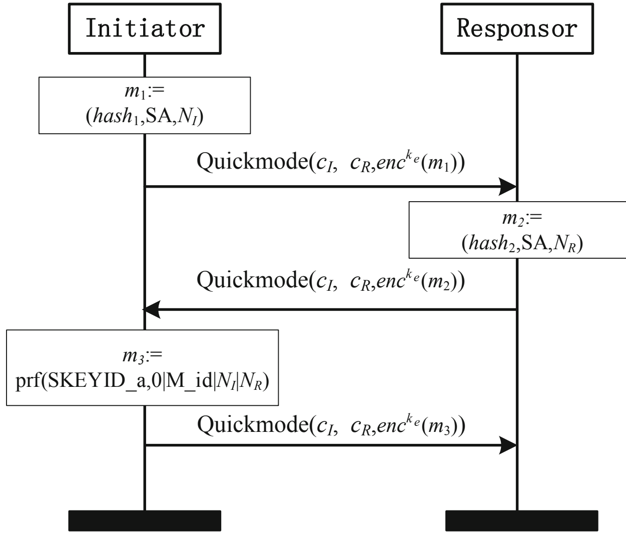

Fig. 2. Generic structure of IKEv1 Phase 1 in quick mode.

$$k_d = prf_k(g^{xy}, c_I, c_R, 0)$$

$$k_a = prf_k(k_d, g^{xy}, c_I, c_R, 1)$$

$$k_e = prf_k(k_d, g^{xy}, c_I, c_R, 2)$$

Signature-Based Authentication. Signature-based verification requires both parties to provide valid certificates for verification. Both parties' certificates and the authentication information will be sent to each other in message 5 and message 6 in the main mode. The key calculation method is as follow:

$$k = prf_{nonce_I, nonce_R}(g^{xy})$$

$$k_d = prf_k(g^{xy}, c_I, c_R, 0)$$

$$k_a = prf_k(k_d, g^{xy}, c_I, c_R, 1)$$

$$k_e = prf_k(k_d, g^{xy}, c_I, c_R, 2)$$

2.2 Fuzzing

In recent years, fuzzing has been widely used in the discovery of software vulnerabilities due to its conceptual simplicity and low deployment barrier [6]. Fuzzing test technology

refers to the generation of test cases with certain rules and inputting them into the program under test. These inputs may be grammatically or semantically wrong, that is, using "fuzzing input" to run the program under test (PUT). There are many tools [7–10] used in software testing and detecting bugs and security vulnerabilities successfully.

As a dynamic testing method based on random data feedback scheme, fuzzing was first pioneered by Miller et al. [11]. Although the fuzzing technology is simple, it is currently one of the most effective methods for security testing due to its effectiveness and high degree of automation, and it is widely used in the vulnerability mining.According to the semantic granularity observed by the fuzzer in the fuzzing test, the fuzzers can be divided into three groups: black-box, gray-box and white-box fuzzers [6]. Gray-box fuzzing is a variant of white-box fuzzing.

Black-Box Fuzzer. Black-box testing [12] is usually used when the observer cannot see the inside of the PUT. We can only treat it as a black-box by observing the input/output. However, it is difficult to construct effective malformed data, so the code coverage of this technology is relatively low.

White-Box Fuzzer. Unlike black-box, white-box fuzz [13] testing needs to analyze the internal structure of the PUT and record the internal execution of the PUT at the same time.

Gray-Box Fuzzer. Gray-box [14] testing is a middle -ground approach between black box testing and white box testing. That is, part of the information of PUT execution can be obtained, however, there is no need to analyze the complete semantics of PUT.

The boundaries of the three are not very clear. In order to generate effective testcases, the black-box fuzzer will also collect some information, and the white-box fuzzer will also make some approximations to reduce complexity [6].

Differential Fuzzing. Differential fuzzing [15] is a technique that provides the same input to different but similar implementations to focus on the behavior/response of the SUT. Different from gray-box technology where code coverage is known, differential fuzzing uses the behavioral asymmetries between multiple test programs as a guide, and can find more semantic errors, which has its unique advantages in the discovery of protocol vulnerabilities.

3 Related Work

It is very important to find security vulnerabilities in protocol implementation, but there are also many challenges. In recent years, there are many learning attempts to apply fuzzing test to the discovery of network security protocol vulnerabilities.

In 2014, Sounthiraraj et al. [16] developed the SMV-HUNTER and used the idea of combining dynamic and static to detect SSL/TLS certificate verification vulnerabilities in multiple Android apps. In the same year, Brubaker et al. [17] fuzzed the SSL/TLS server certificate authentication module using forged certificates. In 2015, Gascon et al. [18] designed PULSAR system based on private protocol fuzzing, which extended the

protocol reverse to the field of fuzzing, attempting to discover the hidden vulnerabilities of protocol systems. In 2016, Somorovsky [19] presented a TLS protocol analysis tool TLS-Attcker and proposed a two-stage fuzzy method to evaluate TLS server behavior.

In 2017, a domain-independent differential testing tool NEZHA was presented by Petsios [20], which uses the asymmetry of behavior between different programs to focus on triggering more semantic errors. In the same year, Walz [21] et al. proposed a new concept of Generic Message Trees to generate TLS fuzzy messages, and analyzed the implementation of TLS protocol using black-box differential fuzzing. In 2020, Reen [22] proposed the differential fuzzing tool DPIFuzz, a structure-aware and modular fuzzing framework which allows automated testing of QUIC implementations by generating and mutating communication streams and a differential analysis of the behaviour of the implementations to these communication streams. In the same year, Phamand et al. [23] designed a fuzzer called AFLNET, the first gray-box fuzzer for protocol implementation. AFLNET makes automated state model inferencing and coverage guided fuzzing work hand in hand and uses state-feedback to guide the fuzzing process. No protocol specification or message grammars are required.

According to the characteristics of the protocol, the most popular technology at present is state based black-box fuzzing, such as Sulley [24], Peach [25]. The key idea of these fuzzers is to test the target by roughly traversing the state machine model of a given system and generating effective message sequences using corresponding syntax. AFLNET was presented as the first gray-box fuzzer, which can automatically infer the state model and use code coverage to guide fuzziness. Another fuzzing method used is the protocol state fuzzing proposed by de Ruiter et al. [26], which infers the state machine of the target protocol by sending regular packets.

However, the existing fuzzing algorithms still have many challenges: Complex protocols (such as IPsec) have huge state space and complex encryption algorithms, so it is difficult to carry out fast and effective fuzzing, and the algorithm is difficult to cover the whole space; The high correlation between protocol data streams also brings challenges to fuzzing test case generation.

4 Method and System Design

4.1 Fuzzer Design Overview

To solve the above problems, we designed the fuzzer as follows (Fig. 3):

The figure above shows the architecture of our fuzzer, composed of sequence generator, packet generator, mutator, encryptor, and response analyzer.

Sequence Generator: Inspired by the protocol state fuzzing, we generate message sequences randomly. Under this module, we can generate various stream sequences and discover the security problems caused by the combination of multiple messages. Note that what is produced here is not a real packet, but an abstract representation.
Packet Generator: This module can convert the message sequence to a real packet. The subsequent packets generation will use the parameters of the previous packet. In order to ensure the correct interaction with the server, the response of the server needs

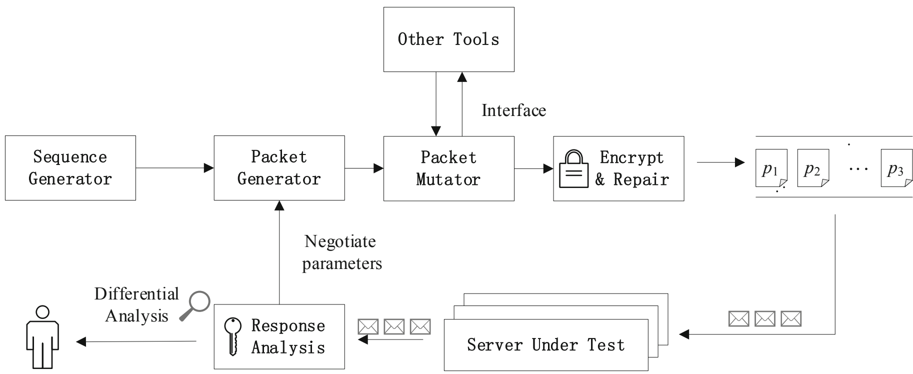

Fig. 3. The overview of our fuzzer.

to be input to the packet generator, that is, the generation of the packet is a process of continuous interaction.

Mutator: Every packet in a sequence can undergo packet mutation a certain probability. There are some fuzzing operators in our mutator. The packet-level fuzzing is mainly aimed at fields.

Encryptor: We set this module to encrypt (optional) and repair (optional) mutated packets. We assume that we are a legitimate client when performing fuzzing. The repair module is mainly to maintain the internal consistency of the packets due to the mutation, which could avoid making the packet rejected by the server. Of course, for the diversity of data packets, the repair operations are optional.

Response Analyzer: This module is mainly used to decrypt and analyze the received packets, record inconsistent replies and abnormal replies. At the same time, there is a connector between the response analyzer and the packet generator to transmit the necessary parameters.

4.2 Mutation and Encryption

4.2.1 Mutation

Different from the traditional fuzzing method, we need to use the semantics and format of IPsec. First, we select the fields to be mutated randomly, and then fuzzing them. We mainly consider the following six fuzzing operations:

Remove operator Randomly delete some bytes for the selected field.
Insert operator Insert a random payload at a random position in the field.
Duplicate operator Copy the selected field and insert it after the field.
Byte operator Randomly obfuscate some of the bytes in the selected field.
Void operator Sets the field to the string of length zero.

4.2.2 Certificate Verification

Authentication of user identity through digital certificate is a very common method, which is very important for network security. For the protocol implementation, certificate

verification is a complex process. X.509 certificate has extremely complex structure and strict semantic syntax. Any direct changes to the certificate will result in validation failure, and the probability of randomly generated certificate will be directly rejected. In recent years, there have also been a lot of work related to certificate verification, such as Frankenert [17], Mucert [27] NEZHA [20] HVlearn [28], etc.

For the fuzzing test of a certificate, we mainly focus on the tbscertificate field in the certificate body, mainly including serial num, issuer, validity, subject and extension fields. Unlike frankenert, which uses a large number of seed certificates, we use the fuzzing operator to make changes on the original certificate, and use the CA's private key to sign the certificate.

4.2.3 Encryption and Repair

After a series of mutations, the frame structure of the packet will change greatly. During server authentication, some messages will be rejected prematurely. In order to avoid message inconsistency, we need to repairing the length field optionally.

Since the IPsec packet contains multiple loads, referring to Walz's [21] packet repair method, considering the trade-off between generating inconsistent packets and repairing inconsistent packets, we can selectively correct each part of the length field with a probability of correction is $\frac{1}{2}$.

IPsec enables encryption from the fifth packet in the main mode, and will provides a large number of optional encryption schemes. In order to ensure the integrity of the message, we adopt the method of mutation before encryption.

The packet processing algorithm is as follows:

Algorithm 1 The packet processing algorithm

```
1:  procedure MUTATE (pck, encrypt)
2:      operator = RANDOMCHOICE ({REMOVE, INSERT, ... ,VOID, NONE})
3:      field = CHOICEFIELD(pck)
4:      mutpck = OPERATE(pck, field, operator)
5:      need_repair = RANDOMCHOICE ({True, False})
6:      if need_repair == True:
7:          mutpck=REPAIR(mutpck)
8:      end if
9:      if encrypt == True:
10:         mutpck = ENCRYPT(mutpck)
11:     end if
12:     return mutpck
13: end procedure
```

4.3 Sequence Generation

Traditional fuzzing method usually produces the correct stream message sequence purposefully. For example, when we are fuzzing the first packet in the quick mode, we need

to provide the correct message sequence to enable the server to respond normally in the main mode. However, this method cannot reveal the relationship between different messages.

Inspired by the protocol state fuzzing, we add the sequence fuzzing at the same time. However, we have no way to perform equivalent queries, nor can we guarantee the closeness and consistency of the algorithm. Therefore, we introduce the length parameter l to control the length of the generated flow. l indicates the maximum number of repetitions of data packets. For example, if l takes 2, the frequency of all data packets in a stream shall not exceed 2. This kind of operation can not only disrupt the sequence, but also make the resulting data flow sequence do not extend indefinitely.

The sequence generation algorithm is as follows:

Algorithm 2 The sequence generation algorithm

1: **procedure** SEQGEN($\mathcal{I}$, l)
2: $counter = \{\}$, $seq = \{\}$, $v =$ True
3: **for** msg **in** $\mathcal{I}$ **do**
4: $\quad counter[msg] = 0$
5: **end for**
6: **while** v **do**
7: $\quad msg =$ RANDOMCHOICE($\mathcal{I}$)
8: $\quad counter[msg] = counter[msg] + l$
9: $\quad seq \cup = \{msg\}$
10: $\quad$ **if** $counter[msg] > l$ **do**
11: $\qquad \mathcal{I} = \mathcal{I} \setminus msg$
12: $\quad$ **end if**
13: $\quad v =$ RANDOMCHOICE({True, False})
14: **end while**
15: **return** seq
16: **end proceduce**

5 Result

In this section we present our experiment on IPsec implementation. We use the method above to analyze the target protocol and find the differences between the protocols. In order to ensure that our tool can communicate with the target server normally, we need to generate separate sessions for each server, record the connection parameters of each session, and ensure that only the tested field will be mutated. We implement our entire framework using python. For our experiments, we consider five open source implementations of IPsec, including strongSwan 5.3.4, strongSwan 5.9.2, Libreswan 3.0 and Libreswan 4.4.

5.1 Careless Field Checking

We found that the four similar protocol implementations have some fields that are checked carelessly, resulting in inconsistent field phasing, which shows the difficulty of protocol implementation. Although the problematic field will not have a substantial impact on the protocol security, the error handling and inconsistent resolution of the length field may lead to the server's incorrect resolution of the protocol message.

1) *Libreswan: Fields Lack of Checking*
   An ISAKMP message has a fixed header format. However, some implementations lack checks on these fixed messages. For example, the Proposal Transforms field specifies the number of transforms for the Proposal, but Libreswan is lack of checking. We can pass the verification by replacing it with any octet. The default setting for the reserved field is 0. The Libreswan does not check this field, but strongSwan checks it.
2) *StrongSwan: Fields Lack of Checking*
   In particular, RFC stipulates that the requests for assignment of new life types MUST be accompanied by a detailed description of the units of this type and its expiry. However, strongSwan lacks sufficient checks on IKE attribute. Life-type with unknown type and life-duration with value 0 can also pass the verification of the server.

5.2 Certificate Verification

Protocol certificate verification is a very complex process. As the key step of identity authentication, improper and lax handling of any part may lead to serious security problems. Here are strongSwan's problems in certificate processing:

1) *Host name validation is not case sensitive*
   During the fuzzing, we found that strongSwan is not case sensitive in host name verification. This case insensitive verification of public name will lead to identity counterfeiting by malicious clients of the internal users.
2) *Overlong Serial number*
   In RFC, the serial number can be a long integer. Certificate users must be able to handle serial number of up to 20 octets. StrongSwan allows serial number certificates greater than 20 bytes.
3) *Mishandling Certificate extension verification*
   A certificate-using system MUST reject the certificate if it encounters a critical extension that it does not recognize or a critical extension that contains information that it cannot process. If there is an extended Key Usage extension, users must check the Key Usage to verify that the certificate is authorized for its purpose. strongSwan 5.3.4 does not check the use of keys and alternate names, which was fixed in 5.9.2.

   A non-critical extension MAY be ignored if it is not recognized, but MUST be processed if it is recognized. Given a certificate with a known non critical extension, but not a valid value of the extension, and strongSwan chooses to accept the certificate.

6 Prospects and Future Work

In this paper, we propose a differential fuzzing framework for IPsec, which uses stream fuzzing and packet mutation to generate IPsec message streams. Compared with the previous fuzzing methods, our method can discover the difference responses between different protocols implementations effectively. The difference in the implementation of the protocol is mainly due to the large parameter space of the protocol specification and the inaccuracy of some specifications. These flexibilities will cause difficulties for experimenters. Our method can cover the main state space of the protocol implementation, which is an effective attempt to apply fuzzing to large-scale cryptographic protocols.

We also plan to further expand our work. First, we hope to add support for other cryptographic protocols, such as IKEv2, SSH, etc. In addition, our packet generation process did not use the feedback of the message. In the next step, we will enrich our mutation strategy and borrow gray box technology to optimize our packet generation process. Finally, finding multi trigger security vulnerabilities through flow ambiguity testing is another key idea. We plan to study this potential in more detail.

Acknowledgments. This work has been supported by the National Natural Science Foundation of China under Grant 61772548.

References

1. “OpenSSL ‘Heartbleed’ vulnerability”. CVE-2014–0160 (2013)
2. Kent, S., Seo, K.: Security architecture for the internet protocol. RFC 4301 (Proposed Standard) (2005). http://www.ietf.org/rfc/rfc4301.txt
3. Harkins, D., Carrel, D.: The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard) (1998). http://www.ietf.org/rfc/rfc2409.txt (obsoleted by RFC 4306. Updated by RFC 4109)
4. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P.: RFC 5996: Internet Key Exchange Protocol Version 2 (IKEv2) (2010). http://www.rfc-editor.org/info/rfc5996
5. Felsch, D., Grothe, M., Schwenk, J., Czubak, A., Szymanek, M.: The dangers of key reuse: practical attacks on IPsec IKE. In: Proceedings of the 27th USENIX Conference on Security Symposium (SEC 2018), pp. 567–583. USENIX Association, USA (2018)
6. Manes, V., Han, H.S., Han, C., et al.: Fuzzing: art, science, and engineering (2018)
7. Google. Honggfuzz: security oriented software fuzzer (2015). https://github.com/google/honggfuzz
8. LLVM. libFuzzer: a library for coverage-guided fuzz testing (2015). http://llvm.org/docs/LibFuzzer.html
9. Zalewski, M.: American fuzzy lop (n. d.). http://lcamtuf.coredump.cx/afl/
10. TLS-Attacker: a Java-based framework for analyzing TLS libraries. https://github.com/RUB-NDS/TLS-Attacker
11. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
12. Beizer, B.: Black-box Testing: Techniques for Functional Testing of Software and Systems. Wiley, Hoboken (1995)
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, pp. 151–166 (2008)

14. DeMott, J.D., Enbody, R.J., Punch, W.F.: Revolutionizing the field of grey-box attack surface testing with evolutionary (2007)
15. McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998)
16. Sounthiraraj, D., Sahs, J., Greenwood, G., et al.: SMV-HUNTER: large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium. NDSS 2014 (2014)
17. Brubaker, C., Jana, S., Ray, B., et al.: Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In: 2014 IEEE Symposium on Security and Privacy, pp. 114–129. IEEE (2014)
18. Gascon, H., Wressnegger, C., Yamaguchi, F., et al.: PULSAR: stateful black-box fuzzing of proprietary network protocols. In: Thuraisingham, B., Wang, X., Yegneswaran, V. (eds.) International Conference on Security and Privacy in Communication Systems. Springer, Cham, pp. 330–347 (2015). https://doi.org/10.1007/978-3-319-28865-9_18
19. Somorovsky, J.: Systematic fuzzing and testing of TLS libraries. In: Proceedings 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016), pp. 1492–1504 (2016)
20. Petsios, T., Tang, A., Stolfo, S.J., Keromytis, A.D., Jana, S.: NEZHA: efficient domain-independent differential testing. In: Proceedings of the 38th IEEE Symposium on Security & Privacy, San Jose, CA, May 2017 (2017)
21. Walz, A., Sikora, A.: Exploiting dissent: towards fuzzing-based differential black-box testing of TLS implementations. IEEE Trans. Dependable Secur. Comput. 1 (2017). https://doi.org/10.1109/TDSC.2017.2763947
22. Reen, G.S., Rossow, C.: DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In: Annual Computer Security Applications Conference (ACSAC 2020), pp. 332–344. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3427228.3427662
23. Roychoudhury, A., Pham, V.-T., Böhme, M.: AFLNET: a grey-box fuzzer for network protocols. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE (2020)
24. Amini, P., Portnoy, A., Sears, R.: Sulley (n. d.). https://github.com/OpenRCE/sulley
25. Eddington, M.: Peach fuzzing platform (n. d.). http://peachfuzzer.com
26. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: Proceedings 24th USENIX Security Symposium (USENIX Security 15), pp. 193–206 (2015)
27. Chen, Y., Su, Z.: Guided differential testing of certificate validation in SSL/TLS implementations. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE), pp. 793–804. ACM (2015)
28. Sivakorn, S., Argyros, G., Pei, K., Keromytis, A. D., Jana, S.: HVLearn: auto mated black-box analysis of hostname verification in SSL/TLS implementations (2017)