Online Model-Based Behavioral Fuzzing

Martin Schneider, Jürgen Großmann,
Ina Schieferdecker
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
Email: {martin.schneider, juergen.grossmann,
ina.schieferdecker}@fokus.fraunhofer.de

Andrej Pietschker
Giesecke & Devrient GmbH
Prinzregentenstr. 159
81677 Munich, Germany
Email: andrej.pietschker@gi-de.com

***Abstract*—Fuzz testing or fuzzing is interface robustness testing by stressing the interface of a system under test (SUT) with invalid input data. It aims at finding security-relevant weaknesses in the implementation that may result in a crash of the system-under-test or anomalous behavior. Fuzzing means sending invalid input data to the SUT, the input space is usually huge. This is also true for behavioral fuzzing where invalid message sequences are submitted to the SUT.**

Because systems getting more and more complex, testing a single invalid message sequence becomes more and more time consuming due to startup and initialization of the SUT. We present an approach to make the test execution for behavioral fuzz testing more efficient by generating test cases at runtime instead of before execution, focusing on interesting regions of a message sequence based on a previously conducted risk analysis and reducing the test space by integrating already retrieved test results in the test generation process.

***Keywords*—Model-based Testing; Security Testing; Test Generation; Test Execution; Behavioral Fuzzing**

I. Introduction

Fuzz testing or fuzzing is a security testing approach where invalid input data are sent to the SUT. The goal is to find bugs in the implementation of the SUT that lead to processing instead of rejecting invalid input data. This way, weaknesses in the implementation may be revealed that might be exploited in order to crash the SUT or to modify its behavior in an unintended way. Because the interface of the SUT is stressed with invalid input data, fuzzing is a kind of interface robustness testing and negative testing.

Fuzzing dated back from Barton Miller who used UNIX command line utilities over a noisy dial-up line ([1], foreword by Barton Miller). Due to that noise, spurious characters were induced that caused the command line utilities to crash. A systematic investigation [2] revealed that software is generally susceptible to invalid inputs that lead to security-relevant weaknesses. The kind of fuzzing that was originally conducted by Miller is referred to as random fuzzing because the input data is randomly generated without protocol knowledge. While this approach is even today a way to find vulnerabilities in current software [3], [4], [5], random fuzzing suffers from a number of disadvantages. One is caused by its randomness: when input data is randomly generated, it is usually invalid ([1], p. 27). This means it contains invalid input data in all or nearly all parts. Such totally invalid input data is usually easy to detect and rejected by a SUT even if it is faulty because if one input validation mechanism fails, another may detect another invalid part of the input. Hence, the chance to detect errors in the input validation mechanism of the SUT's interface that lead to processing invalid input data is very small [6]. Another disadvantage results from the lack of protocol knowledge that is used to communicate with the SUT. This leads to input data that is in fact able to find simple bugs. But there is only little chance to reveal bugs that are hidden by complex interaction with the SUT. For instance, generating a bunch of valid messages and then sending a message that carries invalid input data is less likely to be created by a random-based fuzzer than by fuzzers with protocol knowledge. Hence, random-based fuzzers mostly find just simple bugs ([1], p. 27).

Therefore, more sophisticated fuzzers have protocol knowledge in order to generate input data that is not totally invalid but only invalid in small parts. Such input data is called semi-valid because it is mostly valid and invalid only in those small parts. Depending of the amount of protocol knowledge, different categories of fuzzers can be distinguished. *Block-based fuzzers* split messages into static and dynamic parts where only the dynamic parts get fuzzed. *Model-based fuzzers* or *smart fuzzers* have a complete model of the SUT's protocol and hence, are able to find more complex bugs by creating complex interactions with the SUT.

While fuzzing of input data is an already well-established approach for security testing, behavioral fuzzing is quite new. It consists of generating invalid message sequences instead of invalid input data and hence, generally leads to test cases that are as long as or even longer than valid message sequences. As protocols are getting more complex, such message sequences grow further and, with it, its execution time. This is a serious problem if the initialization of the SUT as well as the test execution itself is time-consuming.

We refer to test case generation before execution as offline generation and in contrast to test case generation at test runtime as online generation. Online generation of test cases has two advantages: it avoids generation of all test cases before execution and thus, reduces resources required to generate a possibly large number of test cases before the first one is executed. On the other hand enables online generation of test cases to include results from already executed test cases to guide the test case generation. This allows to achieve certain goals dynamically, e.g. a certain code coverage that is measured at test runtime or to avoid generating test cases which all reveal the same fault

because they contain a common preamble.

In this paper, we present a way to reduce the test execution time by (a) generating test cases by employing model-based behavioral fuzzing [7] stepwise during test execution, (b) respecting the results from the previous test executions for generation of further test cases, and (c) focusing on message subsequences based on a previous conducted risk analysis. The rest of this paper is organized as follows: Section 2 introduces a motivating case study from the banking domain provided by Giesecke & Devrient for the ITEA-2 research project DIAMONDS, in Section 3 we explain in short the behavioral fuzzing approach applied to the case study. The approach for combining test generation and execution including an algorithm is presented in Section 4. Section 5 discusses related work regarding behavioral fuzzing. The paper closes with conclusions and future work.

II. Motivating Case Study

The motivation for the idea of combining test case generation and execution came from applying behavioral fuzzing to a case study from the ITEA-2 research project DIAMONDS. It is a banknote processing machine from Giesecke & Devrient that is able to sort banknotes by its currency, denomination, condition and authenticity. Banknotes run through the machine and are scanned by sensors. The sensor data is evaluated by the software and causes the correct sorting of the banknotes. The software is running on a Machine PC that is part of the banknote processing machine. Giesecke & Devrient provides the software on the machine PC as a virtual machine and additionally a set of functional test cases. These test cases constitute our basis for model-based security testing of the software.

On an abstract level, using the banknote processing system consists of four steps. At the beginning, an operator has to authenticate himself in order to get access to the machine. In the second step, the operator configures the machine for counting certain kinds of banknotes, e.g. by selecting the currency and the denomination of the banknotes. After that, he starts the counting process and the machine automatically counts all the banknotes. This process takes several minutes. If that is done, the operator performs a logout from the machine. Before a functional test case is started, the SUT is initialized by starting the virtual machine using a certain snapshot that ensures a consistent state.

The execution of a functional test case takes considerable time, in our test bed at Fraunhofer FOKUS about 9 minutes. Figure 1 illustrates how execution time is distributed to the different phases of a functional test case. Login and logout are instantaneous events in time after the startup phase and at the end of the test case. The phase *startup* includes starting the virtual machine, its operating system and the software of the Machine PC. The phases *startup* and *counting* (white blocks in Figure 1) run automatically where no messages are received from or sent to the SUT. Only the gray and black phases (*login*, *logout*, and *configuration*) are the focus of these security-related tests. Obviously, only little execution time is useful for testing (about 1/6 of the whole execution time). Therefore, our goal is to avoid or reduce the useless (white) phases that are not usable for security testing and to keep the system in the black and gray phases. In order to achieve this goal, we could omit the counting phase, should reduce the number of startup phases and remain in the black or gray phases (*login* and *configuration*).

III. Model-based Behavioral Fuzzing

Behavioral fuzzing is a security testing approach complementary to traditional data fuzzing. Data fuzzing consists of sending invalid input data in order to reveal weaknesses due to faulty input validation mechanism. In contrast, behavioral fuzzing consists of submitting invalid message sequences to a system under test in order to reveal weaknesses in the implementation. In contrast to most behavioral fuzzing approaches that use finite state machines for test case generation, we generate behavioral fuzz test cases from UML sequence diagrams (as explained in detail in [7]). We apply a set of fuzzing operators to a valid UML sequence diagram. The result is an invalid message sequence that represents a behavioral fuzz test case. The fuzzing operators can be divided in six categories based on their usage of protocol knowledge and the generated deviation to a valid sequence [7]. A fuzzing operator modifies one or more elements of a sequence diagram, e.g. a single message, a set of messages or the guard of a combined fragment. Examples of fuzzing operators are *Remove Message*, *Move Message*, *Negate Interaction Constraint*, and *Change Bounds of Loop*. A complete list can be found in [7].

Applying behavioral fuzzing to UML sequence diagrams instead of to state machines has the advantage that the approach could be applied even if no state machine is available. Behavioral fuzzing of UML sequence diagrams allows using other information, e.g. using a general conformance test suite for a protocol or functional test cases developed for a certain SUT as in our case. We use the functional test cases of the case study as a basis for our behavioral fuzz tests.

Giesecke & Devrient provided the case study by delivering a virtual machine of the banknote processing system as well as a set of functional test cases written in TTCN-3. Behavioral fuzzing of UML sequence diagrams allows us to re-use their functional test cases for security testing. In order to apply model-based behavioral fuzzing, we first generated a model of these functional test cases using sequence diagrams. In a second step, we applied our behavior fuzz test case generator to these functional test cases in order to generate behavioral fuzz test cases.

IV. Online Model-based Behavioral Fuzzing

As discussed in [7], the number of security test cases that can be generated by applying behavioral fuzzing to a single UML sequence diagram can be approximated by

$$\mathcal{O}(\sum_{i=1}^{n} \frac{o'!}{(o'-i)!}) \tag{1}$$

with

$$o' = o \cdot e^k \tag{2}$$

The number of available fuzzing operators that can be applied to a sequence diagram is denoted by o. e is the number of elements in the sequence diagram that can be fuzzed.

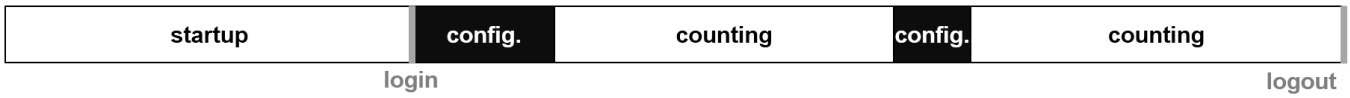

Fig. 1. Execution Time of a Functional Test Case

Depending on the applicable fuzzing operators, this could be the number of messages, or the number of messages plus the number of guards within interaction operands, for instance. n is the maximal number of fuzzing operators to be applied to a sequence diagram to generate a single test case. k is a constant representing the number of different modifications that can be applied to an element by a single fuzzing operator. E.g., k is 1 for fuzzing operators that remove or repeat a message, and 2 for fuzzing operators that move messages.

Obviously, the number of test cases is too large to execute them all, especially when considering systems as the banknote processing machine presented in Section II.

Traditional test case generation before execution is not always sufficient. On one hand, when a weakness is found while executing a test case, this is mostly not the case after a test case is completely executed. Therefore, the message subsequence that follows that part of the test case that revealed a weakness was generated and executed gratuitously. On the other hand, when a test case found no weakness, this might be possible when further invalid messages would be sent to the SUT after the sequence of invalid messages. In this case, executing a single test case does not serve the goal to find weaknesses in the implementation of the SUT, and further test cases could be executed without restarting the SUT.

Moreover, the execution of all test cases and hence, their generation before execution is not useful. The reason is that test cases may contain message subsequences of other test cases. If these subsequences already revealed a weakness within the SUT, it does not make sense to execute further test cases that contain these subsequences because it is already known that they will reveal a certain weakness and therefore not find weaknesses related to the unexecuted part of the test case.

In the context of the case study presented in Section II, these are serious disadvantages. Startup for executing test cases that do not reveal weaknesses consumes a lot of time, as well as test cases that contain message sequences that are already known to reveal a weakness. In the following, we present a combined approach that seems to be able to overcome these issues.

A. Online Test Case Generation

One building block of our proposed solution is online test case generation. It means that test generation and execution are taking place at the same time having a mutual impact. In the context of behavioral fuzzing of UML sequence diagrams as presented in [7], this means that fuzzing operators are applied to a valid UML sequence diagram in order to determine the messages to be sent to the SUT.

Our basic idea is to execute test cases as long as the SUT is healthy. Obviously, we don't need to restart our SUT as long as it behaves normally. Thus, we can avoid long startup times. This helps in our case study where startup of the SUT takes considerable time. One benefit of this procedure is that more test cases can be executed in the same time in comparison with offline generation. Another is that more complex weaknesses can be found that may be revealed when long message sequences are executed that consists of more than those of a single test case. When considering our behavioral fuzzing approach and our case study, we reuse functional test cases for non-functional security testing. These functional test cases contain a limited set of functionality (a single user session) that is called when executing the test case. Executing a large number of fuzzed test cases one after another as long as the SUT is (or seems to be) healthy means that several user sessions are executed without restarting the SUT. That way, weaknesses may be revealed that cannot be detected by an invalid message sequence generated by fuzzing and executing a single user session.

We generate a single test case by applying one or more fuzzing operators to a valid sequence diagram [7].

We distinguish fuzzing operators on two levels:

- *Fuzzing operator instances* are fuzzing operators whose parameters are all having assigned values. In case of the fuzzing operator *Remove Message* that means the message to be removed is already determined, and for the fuzzing operator *Move Message* the message to be moved as well as the position where it will be moved to is determined.
- *Fuzzing operators'* parameters don't have any values assigned to them. When speaking about fuzzing operators in the context of a certain sequence diagram, this term comprises all fuzzing operator instances that can be created by assigning all possible values valid for this sequence diagram.

In the first step, all operators are determined that are applicable to a given, valid sequence diagram. This results in a set of fuzzing operators. After that, all instances of fuzzing operators are created by assigning values to the parameters of these fuzzing operators. After that step, a single test case is generated by applying one fuzzing operator instances to the valid sequence diagram. This is done for all fuzzing operator instances. In the next cycle, all combinations of two fuzzing operator instances are applied in order to generate test cases. This is done up to a certain number of fuzzing operator instances generating a single test case. This ensures that the process of test case generation terminates and limits the number of test cases to be generated. In case of offline generation, the executable test code is generated and executed after all behavioral fuzz test cases are generated.

In our online generation approach, we generate test cases on demand. This means, when a test case is requested by the test execution environment, a single test case is generated by applying a certain number of fuzzing operator instances to a valid message sequence. In the next step, the test execution

environment requests one message from the test case generator, submits it to the SUT. After that, the health status of the SUT is determined. In case of fuzzing, this could be done using connectivity checks in order to detect a crash, or monitoring the internal state of the SUT to find more subtle bugs [1].

If the SUT is still healthy after submitting a fuzzed message, the next message is submitted. If no more messages are available for the test case being executed, the next test case is generated and executed. For doing so, we need to bring the SUT in a state that is expected when starting execution of the next test case. Hence, some kind of reset message or a sequence of reset messages is necessary that could be sent to reach such a state. If, for example, a test case expects a system where no user is logged in, and the logout message at the end of behavioral fuzz test case is removed by the fuzzing operator *Remove Message*, in order to bring the SUT in the expected state, *logout* would be an appropriate reset message. Determining an appropriate reset message depends on the SUT's behavior and state after receiving an invalid message.

If the SUT reveals any weakness during the execution of a test case, its execution has to be stopped because the SUT may be in an inconsistent state or simply crashed. First, the already sent message sequence is saved for later analysis. This comprises all messages sent since the startup the SUT. As a result of our approach, this message sequence is usually much longer than a single test case because we restart our SUT only if necessary. In the case of a revealed weakness, our SUT requires to be brought in a consistent state in order to enable execution of further test cases. This means for our case study that the virtual machine that runs the Machine PC software is restarted using a certain snapshot.

The approach of online generation avoids generating and saving all the behavioral fuzz test cases whose number is really huge as shown by Formula 1. Instead, only message sequences that revealed weaknesses are saved for later analysis.

B. Integrating Previous Test Results

After sending a (possibly invalid) message from the test execution environment to the SUT, its impact on the SUT is determined. If the SUT remains healthy, the next message is sent to the SUT as described in the previous section.

If the SUT behaves anomalously, a weakness in the SUT was found. This leads to actions described before: The complete message sequence that was sent to the SUT is persisted for later analysis. If the complete message sequence or the last sent message has caused the anomalous behavior of the SUT, is not clear. A detailed analysis may be necessary to find the causative message sequence. Persisting the complete message sequence enables this analysis.

Generating test cases by increasing the number of fuzzing operator instances to be used for generating a single test case may lead to test cases that contain complete other test cases generated by a smaller number of fuzzing operator instances. Executing these larger message sequences would result in revealing the same weakness that was already revealed by another test case whose message sequence is contained. Hence, generating message sequences within a test case that already revealed weaknesses has to be prevented. This is realized by keeping the message sequence of the current test case up to the last message that revealed the weakness. If a new test case is generated, it is compared with these message sequences in order to avoid generating as part of a new test case. This improves efficiency because this also avoids restarting the SUT when no new weakness was found.

Indeed, this slows the test generation down a little bit. But this is on one hand limited by the number of already revealed weaknesses which would be relatively small. On the other hand, restarting the SUT takes so much time that avoiding it seems to be reasonable.

For enabling the execution of further test cases, it is necessary to get the SUT back in a consistent, healthy state. Hence, if a weakness in the implementation was found, a snapshot of the SUT is restored and it is restarted using it.

C. Focusing on Message Subsequences

The behavioral fuzzing approach presented in [7] fuzzes a complete sequence diagram that represents e.g. a functional test case. However, some parts of the sequence may contain preamble and postamble steps which set the SUT into the correct mode and are not part of the test itself. Moreover, the focus of security tests is more narrowed to potential threats and vulnerabilities that result from a previously conducted risk analysis. For example, a result from a risk analysis could be that the SUT might be vulnerable in a certain state. Therefore, testing for that vulnerability is only useful if the SUT is in that state.

When considering Figure 1, obviously we would like to avoid the time-consuming *counting* phase. This could be achieved by defining a region around all the messages from login and configuration phase. Defining such a region causes only messages within that region to be fuzzed. This has two benefits:

- It may reduce test execution time by defining regions that avoid time-consuming messages. While this reduces the execution time of a test case, some weaknesses would be missed when excluding some messages. Therefore, the definition of regions has to be done by experts.
- Considering Formula 1, using regions has a decreasing impact on e, the number of elements fuzzing operators can be applied to. Thus, the overall number of test cases may be reduced.

Another example is an authentication bypass vulnerability. Testing for that vulnerability is only useful if the SUT is in a state where no user is logged in. The implications depend on the kind of fuzzing that is performed. While this results in a region comprising just the *login* message when performing data fuzzing, for behavioral fuzzing the region of interest comprises the messages *login* and *logout* that should be moved after respectively before messages that require authentication. How such behavioral fuzz tests for certain vulnerabilities can be generated using security annotations is discussed in [8]. Such testing requires to adapt the test generation process presented in [7] by selecting an appropriate subset of fuzzing operators and fuzzing operator instances. For example, fuzzing

operators such as *Repeat Message* do generate sequences which are out of scope and hence, should be left out from the test generation process.

The shorter such message sequences that shall be fuzzed, the larger is the benefit regarding the time necessary to execute all test cases. However, how long such a sequence should be depends strongly on the SUT and the functions that are fuzzed. In order to find authentication bypass vulnerabilities as discussed above, it could be sufficient to define a message sequence comprising a *login* message, a *logout* message and a message that shall be protected by *login* or *logout*. Other kinds of functions to be tested may consist of a larger set of messages, which results in larger message subsequences. Excluding a preamble and a postamble from message sequences reduces the number of messages that should be behavioral fuzzed. This is done for the purpose of focusing on the actual functionality to be security tested and to avoid testing parts of the SUT that may not be security-relevant.

In this paper, we focus on behavioral fuzzing of regions employing all applicable fuzzing operators that modify only messages within that region. Hence, all fuzzing operators are applicable that suffice the following conditions:

- The element (a message, a combined fragment or one of its interaction operands or guards) must be contained by such a region.
- The elements that are affected by applying the fuzzing operator instance are all contained in such a region.

D. Algorithm

Combining these building blocks – online test case generation, integration of previous test results, and focusing on message subsequences – leads to the algorithm in Figure 2 we describe in the following.

The inputs to the algorithms are:

- $region$: The region to be fuzzed that is a subsequence of the sequence diagram. It results from runtime observations by defining regions that avoid long execution times or from a previously conducted risk analysis that identified that region to be susceptible for certain threats.
- All combinations of fuzzing operator instances ($allFuzzOpCombs$) that are applicable to $region$ and affect only messages within it.
- A kind of reset message or a sequence of messages that brings the SUT to a state expected at the beginning of a test case.

The algorithm simply returns a set of message sequences $revSeq$ that revealed a weakness. It works as long as remaining combinations of fuzzing operators are available for generating test cases (line 2). When running the first iteration or after revealing a weakness, the SUT is restarted (line 3). The variable $currSeqMsgs$ stores the messages sent to the SUT since its last startup.

The outer while loop (lines 2 to 27) ensures that each combination of fuzzing operator instances is used for test generation.

The body of the inner while loop (lines 5 to 26) constitutes the whole test generation and execution process. In the first step, the SUT is initialized. This is done by sending all messages until $region$ is reached (line 7). These messages constitute a valid message sequence and thus, it could be assumed that the SUT is in the desired state if all these messages were sent to the SUT. Lines 7 to 11 determine the next combination of fuzzing operator instances (line 8) for generating the next test case by applying them to $region$ (line 9) as discussed in [7]. The resulting fuzzed region is compared with all test cases that revealed a weakness and the result is stored in $avoid$ (line 10). It indicates whether the fuzzed region contains any message sequence that already revealed a weakness (consider that this comprises not the whole message sequence since the last SUT startup but only those messages that are generated by the currently executed test case, i.e. fuzzed region). This is done until such a region was generated or no further fuzzed regions can be generated due to no further fuzzing operator combinations are available (line 11). This seems to be a point where efficiency seems to be relevant. Actually, it depends on how much vulnerabilities are found during security testing. For mature systems, this number could be pretty small and hence, this part does not affect the efficiency of the approach that much. However, using special data structures may improve the performance of this part of the algorithm when testing systems where a large number of vulnerabilities were found. If no further fuzzed regions can be generated (line 12), all message sequences that revealed a weakness are returned (line 13). Otherwise, a new fuzzed region representing a single test case for that region is found and can be sent to the SUT. The combination of fuzzing operators is then applied to $region$ in order to generate an invalid message sequence (line 16).

This is done message-wise (for each-loop in lines 16 to 23). The message is sent to the SUT (line 17) and this message is added to the list of messages sent since the last startup ($currSeqMsgs$, line 18). In the next step, it is checked whether the SUT is healthy (line 19). How this is done depends strongly on the SUT and which monitoring techniques can be used. A few monitoring approaches are given in [1]. If the SUT is not healthy, a message sequence that revealed a weakness was found. The whole message sequence since last startup of the SUT is saved for later analysis (line 20). If the SUT remains healthy, execution proceeds with the next message until all messages of the fuzzed region are sent to the SUT. If no weakness was revealed after all messages were sent, the next combination of fuzzing operator instances is tried by the outer while loop after sending a reset message (line 24). If all combinations are tested, all message sequences that revealed a weakness are returned for analysis (line 27).

V. Related Work

In the following, behavioral fuzzing approaches are presented as well as their integration of a feedback mechanism. While behavioral fuzzing is implicitly performed by several approaches, including random-based fuzzing as discussed in [7] and fuzzing using evolutionary algorithms for learning the model [9] or model inference [10], explicit behavioral fuzzing approaches are seldom.

Becker et al. [11] tested the IPv6 Neighbor Discovery

Input: $region$ (region to be fuzzed)
$allFuzzOpCombs$ (all combinations of fuzzing operator instances applicable to $region$ as a stack)
$resetMsg$ (brings the SUT back to the beginning of or before $region$)
Output: $revSeq$ (all messages sequences that revealed a weakness)

```
1:  revSeq := ∅
2:  while (allFuzzOpCombs ≠ ∅) do
3:    startupSUT()
4:    currSeqMsgs = emptyList
5:    while (SUT is healthy) do
6:      initSUT(region)
7:      repeat
8:        nextComb := allFuzzOpCombs.pop()
9:        fuzzedRegion := nextComb.apply(region)
10:       avoid := ({seq ∈ revSeq |
          fuzzedRegion.contains(seq)} = ∅)
11:     until (not avoid or allFuzzOpCombs = ∅)
12:     if (avoid) then
13:       return revSeq
14:     end if
15:     currRegionMsgs := nextComb.applyTo(region)
16:     for each (msg in region) do
17:       msg.send()
18:       currSeqMsgs.add(msg)
19:       if (not SUT is healthy) then
20:         revSeq = revSeq ∪ {currSeqMsgs}
21:         break
22:       end if
23:     end for
24:     resetMsg.send()
25:   end while
26: end while
27: return revSeq
```

Fig. 2. Algorithm for Online MBBF

Protocol using fuzzing. They employed a finite state machine as a behavioral model. They used a second state machine, called strategical state machine. The different states represent different combination of fuzzing strategies. These fuzzing strategies are equivalent to our fuzzing operators. They comprise beside traditional data fuzzing also behavioral fuzzing strategies by inserting, repeating and dropping messages. They included a feedback mechanism comprising the number of functions (power) as well as the number of different functions (entropy) involved processing a single message, if an error or if a corrupt or delayed message was monitored. Hsu et al. [12] used similar fuzzing strategies that changes the type of a message or reorders messages. However, their feedback mechanism was focused on finding messages that might reveal weakness by improving code coverage and anomalous message (corrupt or delayed), it does not focus on avoiding already sent message sequences. The implementation of a Neighbor Discovery Protocol also consists not of a large number of messages, and their combination of fuzzing strategies seems to comprise at most 2. Hence, they do not have the problem of test execution time and message sequences occurring in different test cases. Their approach does not provide possibilities to focus on message subsequences to be fuzzed. Their goal focuses on reducing the number of test cases by the feedback of the SUT. However, they do not describe their strategy in detail.

Banks et al. [13] presented a tool called SNOOZE that allows developing stateful network protocol fuzzers. It uses an XML-based specification of the protocol that contains a finite state machine for describing the flow of messages. A fault injector component is used to generate messages carrying invalid parameters. SNOOZE provides several primitives, for instance to fuzz several values depending on their type. It provides also primitives to get invalid messages and thus, enables development of behavioral fuzzers. However, SNOOZE is a framework for developing a fuzzer and provides means for determining the SUT's state at runtime. Using the information for guiding the test case generation is the responsibility of a developer of fuzzer who uses the SNOOZE framework.

Another approach of behavioral fuzzing is presented by Kitagawa et al. [14]. It focuses on behavioral fuzzing and presented some vulnerabilities found by behavioral fuzzing. These vulnerabilities show that behavioral fuzzing does actually find vulnerabilities that could not be found by employing data fuzzing. They found a vulnerability in Apache web server that results from sending messages of a HTTP GET-request several times when it is allows only once. Thus, an input validation mechanism was bypassed and allowed memory corruption. However, online generation of test cases was not part of their work.

VI. Conclusions and Future Work

We presented an online behavioral fuzzing approach that allows runtime-efficient model-based behavioral fuzzing by avoiding unnecessary restarts of the SUT and reducing the number of different test cases to be executed by focusing on message subsequences defined by regions. These regions may be the results from a previously conducted risk analysis or by observing the execution time of messages and excluding messages that are time-consuming while having little chance of revealing weaknesses. Additionally, results from previously executed test cases are integrated by avoiding subsequences that already revealed a weakness. This also reduces the number of different test cases to be executed.

The approach allows systematic fuzz testing by executing each behavioral fuzz test case in an automated manner. It reduces the test execution time significantly by decreasing the number of time-consuming restarts of the SUT. The number of restarts using online model-based behavioral fuzzing depends only on the number of found weaknesses while offline test generation restarts the SUT for each test case.

However, the approach can be improved. It seems to be crucial to find an appropriate reset message in order to avoid as many restarts of the SUT as possible. As shortly discussed, this could be difficult. After sending message sequence to the SUT that contains a one or more invalid messages, it may be unclear which state the SUT reached. The appropriate reset message or reset message sequence depends on that state. If possible, monitoring the internal state of the SUT would help to determine the state in order to determine a reset message. If this is not possible, finding the reset sequence is more difficult. Additionally, different SUTs could react differently

on an invalid message. While one could ignore this message and would process valid messages after an invalid message, another SUT would reset to an initial state automatically after receiving an invalid message. While the latter case requires no reset message, for the first case the reset message is required to avoid a restart of the SUT.

Another issue is to determine the actual message sequences that revealed a weakness. Our proposed algorithms uses the whole message sequence of a region, in the worst case since startup of the SUT, to determine test cases to be avoided in future executions. In fact, the message sequence could be much shorter. Knowing the actual, shortest message sequence that would reveal a weakness would improve the efficiency by excluding more test cases from execution. An idea to determine this message sequence could be to try out different message sequences from beginning with just the last sent message up to the whole message sequence since last startup. While this would find the shortest message sequence that reveals a weakness, it could also be very time consuming, especially if the shortest message sequence is as long as the initially persisted message sequence.

We are currently implementing this approach for the presented case study by Giesecke & Devrient and look forward to first results.

ACKNOWLEDGMENT

This work was funded by the ITEA-2 research project DIAMONDS. Please see www.itea2-diamonds.org for more information.

The research leading to these results has also received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no 316853.

REFERENCES

[1] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for software security testing and quality assurance*, ser. Artech House information security and privacy series. Artech House, 2008.

[2] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[3] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murphy, A.Natarajan, and J. Steidl, "Fuzz revisited: a re-examination of the reliability of Unix utilities and services," U. Wisconsin, Techn. report CS-TR-95-1268, 1995.

[4] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*. Berkeley, CA, USA: USENIX Association, 2000, pp. 6–6.

[5] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," in *Proceedings of the 1st international workshop on Random testing*, ser. RT '06. New York, NY, USA: ACM, 2006, pp. 46–54.

[6] I. V. Sprundel, "Fuzzing: Breaking software in an automated fashion," *Talk at: 22nd Chaos Communication Congress: Private Investigations*, 2005. [Online]. Available: http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf

[7] M. Schneider, J. Großmann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker, "Behavioral fuzzing operators for UML sequence diagrams," in *7th Workshop on System Analysis and Modelling 2012 (SAMWkshp 2012)*, ser. LNCS, O. Haugen, R. Reed, and R. Gotzhein, Eds., vol. 7744. Springer, 2013, pp. 88–104.

[8] M. Schneider, "Behavioral fuzzing operators for UML sequence diagrams," in *9th Workshop on Systems Testing and Validation, STV'12. Proceedings*, J. Garbajosa, J. Boegh, and A. Rennoch, Eds., vol. 7744. Fraunhofer, 2012, pp. 39–48. [Online]. Available: http://publica.fraunhofer.de/documents/N-217135.html

[9] J. DeMott, R. Enbody, and W. Punch, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," BlackHat and Defcon, 2007. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Whitepaper/bh-usa-07-demott_enbody_and_punch-WP.pdf

[10] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning, "Experiences with model inference assisted fuzzing," in *Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 2:1–2:6.

[11] S. Becker, H. Abdelnur, R. State, and T. Engel, "An autonomic testing framework for IPv6 configuration protocols," in *Mechanisms for Autonomous Management of Networks and Services*, ser. Lecture Notes in Computer Science, B. Stiller and F. De Turck, Eds. Springer Berlin,Heidelberg, 2010, vol. 6155, pp. 65–76.

[12] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," in *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, oct. 2008, pp. 114 –123.

[13] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr," in *Information Security*, ser. Lecture Notes in Computer Science, S. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, Eds. Springer Berlin,Heidelberg, 2006, vol. 4176, pp. 343–358.

[14] 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*, june 2010, pp. 202 –208.