Behavioral Fuzzing Operators for UML Sequence Diagrams

Martin Schneider[1], Jürgen Großmann[1], Nikolay Tcholtchev[1], Ina Schieferdecker[1], and Andrej Pietschker[2]

[1] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
{martin.schneider,juergen.grossmann,nikolay.tcholtchev, ina.schieferdecker}@fokus.fraunhofer.de

[2] Giesecke & Devrient GmbH, Prinzregentenstr. 159, 81677 Munich, Germany
andrej.pietschker@gi-de.com

Abstract. Model-based testing is a recognized method for testing the functionality of a system under test. However, it is not only the functionality of a system that has to be assessed. Also the security aspect has to be tested, especially for systems that provide interfaces to the Internet. In order to find vulnerabilities that could be exploited to break into or to crash a system, fuzzing is an established technique in industry.

Model-based fuzzing complements model-based testing of functionality in order to find vulnerabilities by injecting invalid input data into the system. While it focuses on invalid input data, we present a complementary approach called behavioral fuzzing. Behavioral fuzzing does not inject invalid input data but sends an invalid sequence of messages to the system under test. We start with existing UML sequence diagrams – e.g. functional test cases – and modify them by applying fuzzing operators in order to generate invalid sequences of messages. We present the identified fuzzing operators and propose a classification for them. A description of a case study from the ITEA-2 research project DIAMONDS as well as preliminary results are presented.

Keywords: Model-based Testing, Security Testing, Fuzzing, UML.

1 Introduction

Model-based testing is nowadays a widely used approach for testing the functionality of systems, especially in combination with model-based development. However, the more parts of our daily lives depend on systems the more important is that these systems not just only work correctly but also address adequately various security aspects. The huge number of security incidents shows the great importance of various security aspects and dimensions such as confidentiality, integrity and availability of data and systems. In order to find weaknesses that could be exploited during an attack, fuzzing is an important tool to use. It is a security testing approach that finds vulnerabilities by injecting invalid input data [1]. It aims at finding deviations in the behavior of the system under

© Springer-Verlag Berlin Heidelberg 2013

test (SUT) to its specification which leads to vulnerabilities because invalid input is not rejected but instead processed by the SUT. Such deviations may lead to undefined states of the SUT and can be exploited by an attacker for example to successfully perform a denial-of-service attack because the SUT is crashing or hanging.

The origin of fuzzing dated from Barton Miller, Lars Fredriksen and Bryan So [2]. They injected randomly generated input data into UNIX command line tools and such make them crash. After a first demonstration of this approach for UNIX command line tools in 2000 [2], they showed in 2007 that this approach can be further used for finding vulnerabilities in MacOS [3].

There are different categories of fuzzers:

1. *Random-based fuzzers* generate input data randomly. They know nearly nothing about the protocol of the SUT. Because of the usually huge size of the input space, mostly invalid input data is generated ([4], p. 27).
2. *Template-based fuzzers* use existing, valid traces (e.g. network traces, files) and modify them at some locations to generate invalid input data ([5], p. 49).
3. *Block-based fuzzers* break protocol messages down into static and variable parts and generate fuzzed input data only for the variable parts. They know about field length values and checksums and thus can generate more sophisticated invalid input data ([4], p. 27).
4. *Dynamic generation/evolution-based fuzzers* learn the protocol of the SUT from feeding the SUT with data and interpreting its responses using evolutionary algorithms or active learning [6,7].
5. *Model-based or smart fuzzers* have full knowledge of the protocol used to communicate with the SUT. They use their protocol knowledge to fuzz data only in certain situations that can be reached by simulating the model [8].

Following the traditional approach only input data is fuzzed. Behavioral fuzzing complements this approach by fuzzing not the arguments but the appearance and order of messages. It changes the valid sequence of messages to an invalid sequence by rearranging messages, repeating and dropping them or just changing the type of a message.

Behavior fuzzing differs from mutation testing such that mutation testing in the sense of code fault injection modifies the behavior of the SUT to simulate various situations that are difficult to test ([4], p. 90). Hence mutation testing is a white box approach. In contrast (behavior) fuzzing modifies the use of a SUT such that it is used in an invalid manner. Because the implementation of the SUT does not have to be known behavior fuzzing is a black-box approach.

The motivation for the idea of fuzzing behavior is that vulnerabilities cannot only be revealed when invalid input data is accepted and processed but also when invalid sequences of messages are accepted and processed. A real-world example is given in [9] where a vulnerability in Apache web server was found by repeating the host header in an HTTP request. This vulnerability cannot be found by fuzzing the input data. Data fuzzing would only change the parameter of the host message while behavioral fuzzing would change the number of host messages sent to the web server. Only an invalid number of host messages generated by behavioral fuzzing can reveal this denial of service vulnerability.

2 Related Work

Fuzzing has been a research topic for years. There are several approaches to improve the fuzzing process, in order to generate test data that intrudes deeper in the system under test. The general problem of randomly fuzzed input data is that these data items are largely invalid. Because of that the input data will be rejected by the SUT before getting the chance to get deeper in the SUT [10,11,12]. In that context, model-based fuzzing is a promising approach. Since the protocol is known, model-based fuzzing makes it possible to get deeper in the SUT by fuzzing after passing a certain point in a complex protocol and generating invalid data only for certain message parameters. The model can be created by the system engineer or the tester or it can be inferred by investigating traces or using learning algorithms. There are many possibilities for what can be used as a model. Context-free grammars are widely used as a model for protocol messages [11,13,14]. As a model for the flow of messages, state machines can be employed (as in [10,14,15]) or sequence diagrams as used for the behavioral fuzzing approach presented in this paper.

2.1 Implicit Behavioral Fuzzing

In [11], Viide et al. introduces the idea of inferring a context free grammar from training data that is used for generating fuzzed input data. They used compression algorithms to extract a context free grammar from the training data following the "Minimum Description Length" principle, in order to avoid the expensive task of creating a model of the SUT. The quality of the inferred model directly correlates with the amount and dissimilarity of available traces used for extracting the grammar. Therefore, if the model is not exact, because the available traces have a poor quality, implicit behavioral fuzzing is done when using the inferred model.

Another way of inferring a model of the SUT is by applying evolutionary algorithms. DeMott, Enbody and Punch follow this approach in [6]. They evaluate and evolve pools of sessions, where each session represents a complete transaction with the SUT, using a fitness function that determines the code coverage. The generations are created by crossing pools, selecting, crossing and mutating sessions. After creation of a new generation, the SUT is fed with the sessions of the pools and the fitness of every session and pool is recalculated. This process is stopped after a given number of generations. This is a more advanced but also not explicit way of behavioral fuzzing. Dynamic generation and evolution-based fuzzers try to learn the protocol using different algorithms as mentioned above. At the beginning of the learning process the model is mostly incorrect and so invalid messages and data are sent to the SUT. During the process, the learned model is getting closer to the implemented behavior of the SUT. During this approximation the fuzzing gets less random-based and gets subtler because the difference between the invalid generated behavior and the correct use of the SUT gets smaller. Therefore, implicit behavioral fuzzing performed by dynamic generation and evolution-based is superior to that performed by random-based fuzzers.

But there is a crucial drawback of implicit behavioral fuzzing: While weaknesses like performance degradation and crashes can be found, other kinds of vulnerabilities cannot be detected. That is the case because the revealed behavior of the SUT cannot be compared to a known specification and hence, vulnerabilities, e.g. revealing secret data or enabling code injection, are perceived as intended features.

2.2 Explicit Behavioral Fuzzing

In the PROTOS project on Security Testing of Protocol Implementations [13], Kaksonen, Laakso and Takanen used a Backus-Naur-Form based context-free grammar to describe the message exchange between a client and a server consisting of a request and a response, as well as the syntactical structure of the request and the response messages. The context-free grammar acts as a model of the protocol. In the first step, they replace some rules by explicit valid values. In a second step they insert exceptional elements into the rules of the grammar, e.g. extremely long or invalid field values. In the third step they define test cases by specifying sequences of rules in order to generate test data. Behavioral fuzzing is mentioned in [13] where the application of mutations was not only constrained to the syntax of individual messages but also applied to "the order and the type of messages exchanged" [13]. Understanding behavioral fuzzing in that way, random-based fuzzing implicitly performs behavioral fuzzing. Because the protocol is unknown, randomly generated data can be both messages and data. Hence, in addition to data fuzzing, also behavioral fuzzing is done —- but in a random way.

For testing the IPv6 Neighbor Discovery Protocol, Becker et al. in [15] used a finite state machine as a behavioral model of the protocol and decomposed the messages of the Neighbor Discovery Protocol. They applied several fuzzing strategies, e.g. changing field values or duplicating fields like checksums, which all constitute data fuzzing. The different fuzzing strategies mentioned by the authors are not constrained to fuzzing input data by deleting, inserting or modifying the values of fields but supplemented by the strategies of inserting, repeating and dropping messages which is already to be considered as behavioral fuzzing. Similar strategies are introduced in [7] where the type of individual messages is fuzzed as well as messages are reordered.

Banks et al. describe in [10] a tool called SNOOZE for developing stateful network protocol fuzzers. The tool reads an XML-based protocol specification containing, among other things, the syntax of messages and a state machine representing the flow of messages. A fault injector component allows modifying integer and string fields to generate invalid messages. SNOOZE can be used to develop individual fuzzers and provides several primitives, for instance to fuzz several values depending on their type. Among those primitives, there are functions to get valid messages depending on the state of a session and on the used protocol, but also primitives to get invalid messages. Thus SNOOZE enables fuzzing both, data and behavior.

The most explicit approach of behavioral fuzzing is found in [9]. Kitagawa, Hanaoka and Kono propose to change the order of messages additionally to invalidating the input data to find vulnerabilities. The change of a message depends on a state of a protocol dependent state machine. Unfortunately, they do not describe in which way message order is changed to make it invalid.

3 Fuzzing Operators for UML Sequence Diagrams

The approach of behavioral fuzzing will be presented along UML sequence diagrams. The Unified Modeling Language is a widely used standard to model object-oriented software systems and is currently available in version 2.4.1. It is used to define structural and behavioral aspects of systems. One kind of a behavioral diagram is a sequence diagram. It is a view of an interaction that is used to show sequential processes between two or more objects that use messages to communicate with each other. While in object-oriented programming these messages are method calls, in text oriented protocols such as HTTP they represent specific protocol messages including signaling and payload carrying messages. Messages may have in and outgoing parameters as well as return values. The order of messages represents their appearance in time. Figure 1 (a) shows an example of a sequence diagram.

The goal of behavioral fuzzing of UML sequence diagrams is to generate invalid message sequences. UML sequence diagrams usually show valid message sequences between two or more objects. If we assume that the sequence diagrams define all valid sequences, all other message sequences are invalid. Fuzzing of sequence diagrams generates these invalid message sequences by modifying valid sequence diagrams[1].

We decided on UML sequence diagrams for several reasons: The main reason is that the use of sequence diagrams allows the reuse of functional test cases. While in model-based development often finite state machines are used for describing the behavior of a system, test cases derived from the system's behavior model are generally represented as UML sequence diagrams. By reusing functional test cases generated during model-based testing, non-functional testing of the security aspect could be leveraged. Also existing functional test suites for certain protocols can be reused for non-functional security testing by applying behavioral fuzzing.

[1] In general it is not always practical (and may not even be possible) to define every valid sequence with sequence diagrams, therefore a few fuzzed sequences may on inspection turn out to be valid and should be added to the set of valid sequence diagrams.

3.1 Advantages of UML Sequence Diagrams for Behavioral Fuzzing

Combined Fragments. Since UML 2 sequence diagrams may contain control structures, e.g. loops and alternative branches. These control structures are expressed using *combined fragments*. The semantics of a combined fragment is determined by its *interaction operator*. It consists of that interaction operator that denotes the kind of the combined fragment, e.g. *alternatives*, and one or more *interaction operands* that enclose (e.g. alternative) message sub-sequences. Additionally, each interaction operand may be guarded by a Boolean expression called *interaction constraint*. An interaction constraint has to be evaluated to true so that the message sub-sequence of the guarded interaction operand may be executed.

For example a combined fragment with the interaction operator loop contains exactly one interaction operand. The interaction operand contains an interaction constraint that defines at least a value *minint* that defines the number of executions of the interaction operand. Additionally, it can define an upper bound of executions by defining the *maxint* value in order to specify a range of valid loop iterations. A Boolean expression can be specified that exhibits more constraints under which the interaction operand is executed.

These constraints defined explicitly by interaction constraints as guards for interaction operands and implicitly by the interaction operator and its meaning defined by the UML specification, combined fragments are helpful in generating invalid sequences from valid sequences by violating these constraints.

State Invariants. State invariants are associated with a lifeline of a sequence diagram. They exhibit a constraint that is evaluated during runtime. If the constraint evaluates to true, the sequence is valid, otherwise it is invalid. Thus, violating a state invariant is a way to generate an invalid sequence. However, there are some limitations:

– Because fuzzing is a black box approach, the tested SUT cannot be modified. Hence, the only state invariants that can be violated are those associated with a lifeline that is under control of the test component.
– In addition to constraining the object in a direct way, a state invariant may also refer to a state of a statechart diagram. Because we rely solely on sequence diagrams, we cannot use state invariants that reefer to states of a state machine.

Time and Duration Constraints. Similar to state invariants time and duration constraints are evaluated during runtime and distinguish valid and invalid sequences. Sequences are valid if the time or duration constraint is evaluated to true. Like state invariants time and duration constraints can only be violated if they refer to the lifeline of the test component. But in contrast violating them is easier because the constrained element – time – is in direct control of the test component when sending messages, especially in case of a time limit that must not be exceeded.

3.2 General Approach

The aim of our behavioral fuzzing approach is to generate invalid message sequences by breaking the constraints within valid UML sequence diagrams. In order to achieve that goal, we develop behavioral fuzzing operators. A behavioral fuzzing operator modifies one or more elements of a sequence diagram such that an invalid message sequence is generated. By applying a fuzzing operator to a sequence diagram, another sequence diagram is generated. We benefit from this approach by preserving the possibility to use well developed methods for test case generation from UML sequence diagrams. A second benefit of this approach is that several fuzzing operators can be applied to a sequence diagram one after another, in order to create several invalid parts of a message sequence.

In the following, we will discuss the different kinds of elements of UML sequence diagrams and how their constraints can be broken in order to achieve an invalid sequence. We then use this information to propose a set of behavioral fuzzing operators.

3.3 Fuzzing Behavior of UML Sequence Diagram Model Elements

As discussed above, fuzzing of behavior is realized by modifying the different model elements. This could be done in several ways:

– modifying an individual message,
– changing the order of messages,
– changing combined fragments,
– violating time and duration constraints as well as state invariants if possible.

Messages. Generating an invalid message sequence can be achieved by modifying an individual message. To obtain an invalid sequence diagram, a single message

– can be repeated thus it exists twice (see figure 1(b)),
– can be removed from the sequence diagram (see figure 1(c)),
– can be changed by type that is replacing it by another message,
– can be moved to another position,
– can be inserted.

There are two possibilities of fuzzing two and more messages:

– If two messages are selected these messages can be swapped. One invalid sequence can be generated this way.
– If more than two messages are selected, they can be randomly permuted. Because of its randomness, this approach is less powerful than more directed ones [8].
  A less destructive approach could be rotating the selected messages. It tests stepwise omitting messages in the beginning of a sequence and sending it later.

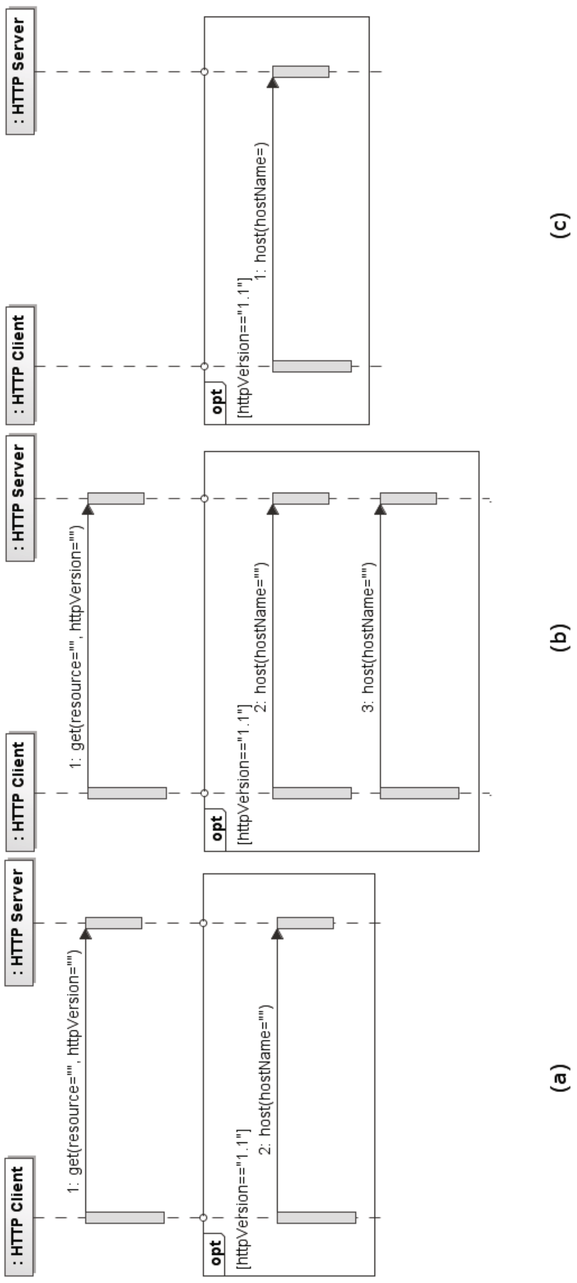

Fig. 1. Modification of an (a) UML sequence diagram (b) after repeating message 2 and (c) after removing message 1

Combined Fragments. Behavioral fuzzing of combined fragments can be done in two ways: considering a combined fragment as a whole or by considering its interaction operands and interaction constraints. When considering a combined fragment as a whole, it can be fuzzed using the same mechanisms as for messages. A combined fragment can

- be removed from the sequence diagram,
- be repeated thus it exists twice,
- be changed by type that means changing its interaction operator,
- be moved to another position,
- be inserted.

The third and the fifth operation - *change interaction operator* and *insert a new combined fragment* - may be difficult to perform. In case of changing a combined fragment's type, meaning to change its interaction operator, depending on the former and the new interaction operator, it is more necessary than just changing the interaction operator. When the former interaction operator is for instance *break* that may have only one interaction operator and is changed to *alternatives* that usually has two interaction operands, a second interaction operand has to be inserted including a message sequence. But filling it with messages in more than a random way is difficult because there are no hints for a certain message sequence that could be inserted in the new interaction operand. This is also true when inserting a new combined fragment. Therefore, removing, repeating and moving combined fragments seem to be the most useful operation when fuzzing combined fragments as whole.

In the following, the different interaction operators defined in the UML Superstructure Specification [16], which seem to be useful for behavioral fuzzing, are discussed.

Alternatives. Combined fragments with the interaction operator *alternatives* realize control structures that are known in programming languages as for instance if ... then ... else ... and switch. They consist of one or more interaction operands each containing an interaction constraint. The interaction constraints of all interaction operands must be disjoint. Hence, at most one interaction operand is executed during one sequence.

To obtain an invalid sequence, the following modifications are possible:

- It is possible to interchange all messages of both interaction operands. This could be done by moving all messages from the first interaction operand to the second and vice versa or by interchanging the interaction constraints of the interaction operands – either in a random manner or rotating them as described for messages above.
- All interaction operands can be merged to a single message sequence and the interaction constraints as well as the enclosing combined fragment can be removed. If there are more than two interaction operands, this could be done by combining stepwise two and more interaction operands until all interaction operands are merged.

Option. An option combined fragment contains an optional message sequence whose execution is guarded by an interaction constraint. It has only one interaction operand. An invalid sequence can be obtained by negating its interaction constraint.

Break. In certain situations, it is necessary to perform some special behavior instead of following the regular sequence diagram. This can be in case of an exceptional situation, for instance if a resource cannot be allocated. To express this in a sequence diagram, the combined fragment with the interaction operator *break* is used. It contains exactly one interaction operand and an interaction constraint. If the interaction constraint is evaluated to true, the interaction operand is executed instead of the remainder of the enclosing interaction fragment.

Invalid sequences can be obtained by the following modifications:

– Negate the interaction constraint. Doing this has the same effect as interchanging the messages of the interaction operand and the remainder of the enclosing interaction fragment.
– Disintegrating the combined fragment results – in contrast to an *option* combined fragment – always to an invalid message sequence because either the interaction operand or the remainder of the enclosing interaction fragment is executed. Disintegrating the combined fragments yields to a sequence where both, the sequence defined by the interaction operand and the remainder of the sequence diagram, is executed.

Weak Sequencing. *Weak* sequencing is the default for how the order of messages must be preserved. If nothing else is specified, weak sequencing is applied to a sequence diagram or an interaction fragment. If weak sequencing is applied, the order of messages regarding each lifeline must be preserved. Moreover, the order of messages that are associated with different lifelines does not need to be preserved. As a consequence, changing the order of messages generates only invalid sequences when they are associated with the same pair of lifelines.

This has an impact on how single or more messages can be modified in order to obtain an invalid message sequence.

Strict Sequencing. In contrast to *weak* sequencing, *strict* sequencing preserves the order of messages independent of the lifelines they are associated with. Thus, invalid sequences can be obtained by changing the order of messages without the necessity to respect the lifelines they are associated with.

Negative. The *negative* combined fragment differs from all other combined fragments in not showing a valid but an invalid message sequence. In order to obtain an invalid sequence, its interaction operator can be changed to *option*.

Consider/Ignore. *Consider* and *ignore* combined fragments are two sides of the same coin. Both are supplemented with a list of messages. In case of an *ignore* combined fragment, it means that these messages are not relevant in order to determine a valid message sequence. Thus, these messages can arbitrarily occur within this combined fragment without affecting the validity of the sequence. In case of a *consider* combined fragment, only the mentioned messages are relevant for a valid sequence. Thus all other messages can arbitrarily occur in the message sequence. Consider and ignore combined fragments cannot behavioral fuzzed itself but has an impact on which of the enclosed messages can be fuzzed.

Loop. A *loop* represents a repetition of a message sequence. It can be set to a certain number of repetitions or limited by a lower and an upper bound. Additionally no limit can be set to tell that all number of repetitions are valid.

Invalid sequences can only be obtained if there is at least one parameter for the loop:

– If there is exactly one parameter, invalid message sequences can be generated by changing it to smaller and greater values to test if there are off-by-one-errors [17].
– If there are two parameters, two different combined fragments can be generated one running from zero to the lower bound -1 and the other by running from the upper bound $+1$ to a maximum number.

Time and Duration Constraints. Time and duration constraints can be used to specify a relative point in time where a message has to be sent or should be received or the amount of time that may elapse between two messages. Time and duration constraints can be given in different situations but only in a few of them can be violated. There are two conditions that must be met to make a violation of a time or a duration constraint possible:

– The constraint has to be associated with a lifeline of the test component. If this is not the case, the constraint has to be maintained by the SUT that is not under control of the test component.
– The occurrence(s), the constraint is associated with, have to be under control of the test component. This is for the same reason as the first condition.

If these conditions are met, the value of the constraint can be negated to generate invalid sequences. The fuzzed constraints must then be respected at test generation time to ensure the original constraint is violated.

State Invariants. State invariants can specify many different constraints on the participants of an interaction, e.g. values of attributes or internal or external states. As for time and duration constraints state invariants has to be under control of the test component. Because of the black box nature of the presented approach only a few of the many different kinds of constraints that can be expressed by a state invariant can be used for fuzzing. These include the valuation of attributes, but not references to external states.

Another challenge when fuzzing state invariants is that just modifying them does not ensure an invalid sequence. If for example the state invariant refers to an attribute of the test component that should have a specific value, by just changing the specified value does not lead to an invalid sequence in terms of the original state invariant. Actually, the behaviors that happen before the state invariant must be changed, in order to achieve the fuzzed state invariant. Additionally the value of the attribute of the test component may be not under immediate control of the test component because the attribute may get its value by the SUT. Thus it is difficult to use state invariants for behavioral fuzzing.

3.4 Summary of Behavioral Fuzzing Operators

Table 1 illustrates the identified behavioral fuzzing operators are illustrated based on the above discussions on the different elements of UML sequence diagrams.

4 Classification Criteria for Fuzzing Operators

In order to generate fuzzed sequence diagrams, one or more of the identified behavioral fuzzing operators can be applied to a sequence diagram that represents valid message sequences. This sequence diagram can originate from a functional test suite. The transformation of sequence diagrams allows the reuse of a functional test suite for non-functional security testing.

Traditional data fuzzing creates a huge number of test cases because the input space is nearly infinite [4], p. 62. That is the reason for the use of heuristics that reduce the size of the input space. The above discussed fuzzing operators are of heuristic nature. When considering the behavioral fuzzing operators, we can estimate how many modifications can be performed for each of them. For the fuzzing operators *remove message* and *repeat message*, one modification per message is possible. For the fuzzing operator *move message*, a message can be moved to the position of each other message that is not enclosed in a negative combined fragment, because that would change the already negative message sub-sequence, or in a *consider/ignore* combined fragment where it is not mentioned respectively mentioned in the list of considered respectively ignored messages. The number of modifications can be estimated by the number of messages enclosed in a sequence diagram. A message can be inserted at the position of each other message where at each position in turn k different messages can be inserted, where k is the number of operations that the class of the lifeline provides, i.e. $|messages|^k$. A first approximation of the number of test cases when applying n behavioral fuzzing operators is given by:

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

with

$$o' = o \cdot e^k \qquad (2)$$

Table 1. Behavioral fuzzing operators for UML sequence diagrams

Operators for...	
Messages	**Constraints**
Remove Message Repeat Message Move Message Change Type of Message Insert Message	- not enclosed in combined fragment *negative* - considered respectively not ignored if enclosed in combined fragment *consider/ignore*
Swap Messages	- not enclosed in combined fragment negative - considered respectively not ignored if enclosed in combined fragment consider/ignore
Permute Messages Regarding a Single SUT Lifeline Rotate Messages Regarding a Single SUT Lifeline	- applicable for messages within a *weak* combined fragment
Permute Messages Regarding several SUT Lifelines Rotate Messages Regarding several SUT Lifelines	- applicable for messages within a *strict* combined fragment
Combined Fragments	**Constraints**
Negate Interaction Constraint	- applicable to combined fragments *option*, *break*, *negative*
Interchange Interaction Constraints	- applicable to combined fragment *alternatives*
Disintegrate Combined Fragment and Distribute Its Messages	- applicable to combined fragments with more than one interaction operand
Change Bounds of Loop	- applicable to combined fragment *loop* with at least one parameter
Insert Combined Fragment Remove Combined Fragment Repeat Combined Fragment Move Combined Fragment	- applicable to all combined fragments except *negative*
Change Interaction Operator	- applicable to all combined fragments
Time/Duration Constraint	**Constraints**
Change Time/Duration Constraint	- applicable to constraints that are on the lifeline of the test component

where o is the number of available fuzzing operators, e is the number of elements in the sequence diagram to be fuzzed, n is the number of fuzzing operators to be applied to a sequence diagram and k is a constant representing the number of different modifications that can be applied to an element by a fuzzing operator. This simple approximation shows that a huge number of test cases can be generated by behavioral fuzzing.

This number of test cases can be too big for executing all possible test cases. Hence, a reasonable classification of the fuzzing operators could be helpful for the test case selection. An attempt of a classification of all behavioral fuzzing operators is illustrated in Table 2.

For traditional data fuzzing, the goal is not to generate totally invalid but semi-valid input data meaning that it is invalid only in a few points. Hence, the number of invalid points generated by a heuristic is of interest. We argue that this is also true for behavioral fuzzing and propose a classification of behavioral fuzzing operators by the number of deviations to the original sequence diagram generated by a fuzzing operator when applied to a sequence diagram.

Another classification criterion could be how the behavioral fuzzing operators relate to random-based or smart fuzzing. As discussed above, model-based or smart fuzzing is more effective than random-based fuzzing because the protocol knowledge is used to generate fuzzed input data. From this point of view, the fuzzing operators that modify a single message or a bunch of messages are rather random-based, because they use only minimal information of the protocol (expressed by a sequence diagram), but modify messages in a random way by e.g. inserting or removing random messages. In contrast, fuzzing operators as for instance *interchange interaction constraints* or *change bounds of loop* use information about the protocol from the sequence diagram and thus, are classified rather to smart fuzzing than to random fuzzing. These classification criteria can be employed to select and prioritize behavioral fuzzing operators for test case generation.

Table 2. Classification of behavioral fuzzing operators

	one deviation	**a few deviations**	**many deviations**
random	- remove message - repeat message - change type of message - insert message	- move message - swap messages	- permute messages regarding a single SUT lifeline - permute messages regarding several SUT lifelines - insert combined fragment
smart	- negate interaction constraint - change bounds of loop - change time/duration constraint	- interchange interaction constraints - disintegrate combined fragment and distribute its messages - change interaction operator - move combined fragment	- remove combined fragment - repeat combined fragment

5 Case Study from the Banking Domain: Banknote Processing System

We developed a prototype implementing 3 behavioral fuzzing operators and applied them to a functional test case from a case study from the banking domain – a banknote processing system. Basically, the banknote processing machine consists of a sets of sensors for detecting banknotes and a PC that analyzes the sensor data in order to determine the currency and the denomination of a banknote, and whether it is genuine or not. In the context of the DIAMONDS research project, we set up the PC with the software at Fraunhofer FOKUS and simulated sensor data of banknotes. The functional test case consists in general of two phases, the first one is the configuration phase where, for instance, the currency and the denomination is selected. The second phase is counting where the (simulated) sensor data is analyzed by the software. We applied the behavioral fuzzing operators *remove message*, *move message* and *repeat message* to that functional test case, in order to generate behavioral fuzzing test cases.

6 Preliminary Results

Due to long execution time of one test case in our setup, we focused on authentication and selected 30 test cases from those generated where the login was omitted or operations (that need login) are performed without any successful login. In that, we could not find any weakness in the software of the banknote processing system.

7 Conclusions and Future Work

We presented a behavioral fuzzing approach working on UML sequence diagrams that was realized by behavioral fuzzing operators. These operators modify a sequence diagram in order to generate an invalid from a valid one. We provided a classification of these operators for test case selection and prioritization. Finally, we presented a case study from the DIAMONDS research project we applied our approach to. We found with a partial prototype implementation no weaknesses in the banknote processing system, but are hopeful to show the efficacy of this approach when more fuzzing operators are implemented in the prototype. We already improved the performance of our test setup that allows us to execute more test cases in future. Additionally, more studies has to be performed on different SUT in order to evaluate the presented approach. Also combination with other kinds of diagram, e.g. statechart diagrams, may help in improving the approach in order to find vulnerabilities.

While the presented approach is applied to UML sequence diagrams, it may be also applicable to message sequence charts (MSCs). MSCs provide similar concepts as UML sequence diagrams, e.g. inline expressions and guarding conditions that are similar to combined fragments and interaction constraints. Further concepts of MSCs, e.g. high-level-MSCs, may be helpful to this approach.

However, there are some issues that must be solved. On one hand, the goal to generate invalid message sequences by applying behavioral fuzzing operators to valid message sequences does not necessarily lead to invalid sequences. That is, in many cases several sequence diagrams specify the protocol of the system under test while a fuzzing operator respects only one sequence diagram when modifying it. Thus, applying a fuzzing operator to one sequence diagram could lead to a message sequence that is specified as a valid one by another sequence diagram and is therefore not invalid in the sense of the specification. By merging several

UML sequence diagram using combined fragments, this drawback may become an advantage because (a) by applying fuzzing operators to a merged sequence diagram, all different message sequences are respected and (b) commonalities and differences that are expressed using combined fragments can be used by the corresponding fuzzing operators that are considered to be more smart by our classification than message-based fuzzing operators.

Other problems are duplicate test cases generated by applying several combination of fuzzing operators that results in the same message sequence, or useless combinations of fuzzing operators such as *move message A* and *remove message A*. We are currently researching efficient ways to overcome these issues. One idea is to ascribe the presented behavioral fuzzing operators to some basic operators, e.g. *insert* and *remove* and to impose conditions on the combinations of these basic operators. That way, useless combinations of operators may be detected and avoided while test case generation.

Another issue is, as always when performing model-based testing, the quality of the model, especially its completeness. The sequence diagrams, e.g. specified for a functional test suite, do not necessarily contain all valid message sequences. Hence, it cannot be ensured that the message sequence generated by behavioral fuzzing operators is an invalid one. This leads to test cases that do not test the security of a system under test while increasing the number of test cases and thus the total test execution time while not revealing any weaknesses.

It is also of interest how traditional data fuzzing and behavioral fuzzing may complement each other, how they can be combined and if this approach is more powerful than applying only one approach at a time.

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

References

1. Bekrar, C., Groz, R., Mounier, L.: Finding Software Vulnerabilities by Smart Fuzzing. In: Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011), pp. 427–430. IEEE Computer Society (2011)
2. Miller, B., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM 33(12), 32–34 (2000)
3. Miller, B.P., Cooksey, G., Moore, F.: An Empirical Study of the Robustness of MacOS Applications Using Random Testing. SIGOPS Operating Systems Review 41(1), 78–86 (2007)
4. Takanen, A., DeMott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, Boston (2008)
5. Uusitalo, I. (ed.): Review of Security Testing Tools. Project deliverable D1.WP3. DIAMONDS Consortium (2011), `http://www.itea2-diamonds.org/_docs/D1_WP3_T1_v11_FINAL_Review_of_Security_Testing_Tools.pdf`
6. DeMott, J., Enbody, R., Punch, W.: Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. Black Hat (2007), `http://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Presentation/bh-usa-07-demott_enbody_and_punch.pdf`

7. Hsu, Y., Shu, G., Lee, D.: A Model-based Approach to Security Flaw Detection of Network Protocol Implementations. In: IEEE International Conference on Network Protocols (ICNP 2008), pp. 114–123. IEEE Conference Publications (2008)
8. Takanen, A.: Fuzzing – the Past, the Present and the Future. In: Actes du 7ème Symposium sur la Séurité des Technologies de l'Information et des Communications (2009), http://actes.sstic.org/SSTIC09/Fuzzing-the_Past-the_Present_and_the_Future/SSTIC09-article-A-Takanen-Fuzzing-the_Past-the_Present_and_the_Future.pdf
9. Kitagawa, T., Hanaoka, M., Kono, K.: AspFuzz: A State-aware Protocol Fuzzer based on Application-layer Protocols. In: IEEE Symposium on Computers and Communications (ISCC 2010), pp. 202–208. IEEE Conference Publications (2010)
10. Banks, G., Cova, M., Felmetsger, V., Almeroth, K.C., Kemmerer, R.A., Vigna, G.: SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 343–358. Springer, Heidelberg (2006)
11. Viide, J., et al.: Experiences with Model Inference Assisted Fuzzing. In: Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies (WOOT 2008). USENIX Association (2008)
12. Forrester, J.E., Miller, B.P.: An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In: Proceedings of the 4th Conference on USENIX Windows Systems Symposium (WSS 2000), p. 6. USENIX Association (2000)
13. Takanen, A., DeMott, J., Miller, C.: Software Security Assessment through Specification Mutations and Fault Injection. In: Communications and Multimedia Security Issues of the New Century, IFIP Advances in Information and Communication Technology, vol. 64, Springer (2001)
14. Abdelnur, H., Festor, O., State, R.: Kif – A Stateful SIP Fuzzer. In: Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm 2007), pp. 47–56. ACM Press (2007)
15. Becker, S., Abdelnur, H., State, R., Engel, T.: An Autonomic Testing Framework for IPv6 Configuration Protocols. In: Stiller, B., De Turck, F. (eds.) AIMS 2010. LNCS, vol. 6155, pp. 65–76. Springer, Heidelberg (2010)
16. Object Management Grouo: Unified Modeling Language, Superstructure v2.4.1, formal/2011-08-06 (2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
17. Denis Howe: off-by-one error. In: Free On-Line Dictionary of Computing, http://foldoc.org/off-by-one+errors