Journal Pre-proof

HFuzz: Towards automatic fuzzing testing of NB-IoT core network protocols implementations

Xinyao Liu, Baojiang Cui, Junsong Fu, Jinxin Ma

PII: S0167-739X(19)32440-9
DOI: https://doi.org/10.1016/j.future.2019.12.032
Reference: FUTURE 5347

To appear in: *Future Generation Computer Systems*

Received date : 14 September 2019
Revised date : 31 October 2019
Accepted date : 25 December 2019

Please cite this article as: X. Liu, B. Cui, J. Fu et al., HFuzz: Towards automatic fuzzing testing of NB-IoT core network protocols implementations, *Future Generation Computer Systems* (2020), doi: https://doi.org/10.1016/j.future.2019.12.032.

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2020) 1–17

Procedia Computer Science

www.elsevier.com/locate/procedia

HFuzz: Towards Automatic Fuzzing Testing of NB-IoT Core Network Protocols Implementations

Xinyao Liu[a], Baojiang Cui*[,a], Junsong Fu[a], Jinxin Ma[b]

[a]*Beijing University of Posts and Telecommunications, No 10, Xitucheng Road, Haidian District, Beijing, China*
[b]*China Information Technology Security Evaluation Center, Building No.1, Courtyard No.8, Shangdixi Road, Haidian District, Beijing, China*

Abstract

Narrowband Internet of Things (NB-loT) is widely deployed in the cellular network of operators, yet implementations of its core network protocols are suffering from bugs. Due to the complexity of the frame structure of NB-IoT core network protocols, testing the protocols in this field is notoriously difficult. In this paper, we propose a novel fuzzing framework, named HFuzz, to generate a great many high-quality test inputs automatically. HFuzz is an automatic hierarchy-aware fuzzing framework and can allocate computing resources efficiently. We put forward the concept of *Message Structure Tree* to transform the seed file and generate mutated data of the tested protocols and optimize the resource allocation for each hierarchy of the transformed structure by a novel scheduling algorithm. Therefore HFuzz can get a balance between breadth and depth in finding new paths. Compared to traditional fuzzing tools, HFuzz can easily pass the early verification and induce a better coverage of the target implementations by taking full advantage of format information of NB-IoT core network protocols. Our framework applies to various protocols, and we evaluate the performance of HFuzz on GPRS Tunnelling Protocol version 2(GTPv2) in this paper and conduct experiments with two protocol implementations, OpenAirInterface(OAI) and B*(a development system). The experimental results show HFuzz yields higher coverage than American Fuzzy Lop (AFL) and Peach, and we further find a real implementation bug in OAI.

Keywords: NB-IoT, core network protocol, fuzzing, hierarchy-aware, complex frame structure

1. INTRODUCTION

In an increasing number of domains, functional safety and user access are significant for secure network communication. It is particularly true in the Internet of Things (IoT), which is a global network of interrelated devices that provide the ability to transfer data. According to the forecast of Internet of Things business, there will be 50 billion IoT connections worldwide in 2020, sixty percent of which will come from Low-power wide-area (LPWA). LPWA network, designed to allow long-range communications at a low bit rate among things, has a number of competing standards and vendors in its space[1] and NB-IoT is one of the most prominent standards, standardized by 3rd Generation Partnership Project (3GPP) for a LPWAN used in cellular networks[2] and evolved from Huawei's Narrowband Cellular IoT (NB-CIoT) effort[3]. Now the Global Mobile Suppliers Association declared that more than 100 operators had adopted either NB-IoT or LTE-Machine-to-Machine (LTE-M) networks[4].

NB-IoT can be deployed directly on Global System for Mobile Communications (GSM) networks[5], Universal Mobile Telecommunications System (UMTS) networks, or Long Term Evolution (LTE) networks[6]. Nowadays, with the wider application of NB-IoT, some automation scenarios, such as intelligent factory, intelligent mine, high-speed railway, etc., deploy NB-IoT by building independent LTE networks to realize connectivity between devices. Some communities also deploy NB-IoT through some open-source LTE architectures such as OAI, Amari, etc. Although 3GPP optimizes the Evolved Packet Core (EPC) architecture of NB-IoT (which we will introduce in Section 2, according to the survey of existing telecom equipment manufacturers, the current equipment of each manufacturer does not use the Cellular Serving Gateway Node (C-SGN) architecture of 3GPP, but still uses the existing EPC equipment architecture. In addition to the application of some security mechanisms[7, 8], security vulnerabilities generally arise from the implementation of the protocols, and we should find the bugs of core network devices before they are commercially available to improve the security of the systems.

In this paper, we would use an automatic fuzzing technique to exploit the vulnerabilities of implementations of NB-IoT core network protocols. Compared to general network protocols, we call NB-IoT core network protocols are the protocols with complex frame structures. To facilitate the description, we first define what the complex frame structure is. If a data block can be divided into header and content, we call it Information Element(IE). As shown in Figure 1, for example, *Recovery* and *Bearer Qos* are both IEs. If there are more than two IEs in the content of any IE in a hierarchy, the protocol has one more layer. In this picture, *Create Session Request*, a type of message of GTPv2, has two hierarchies because its first hierarchy has two IEs: *Recovery* and *Bearer Context*, and an IE of its first hierarchy has two elements: *EBI* and *Bearer QoS*. On account of that, formats of protocols with more than two hierarchies contain more format information, and these protocols are difficult to test. Therefore we define them as protocols with complex frame structures.

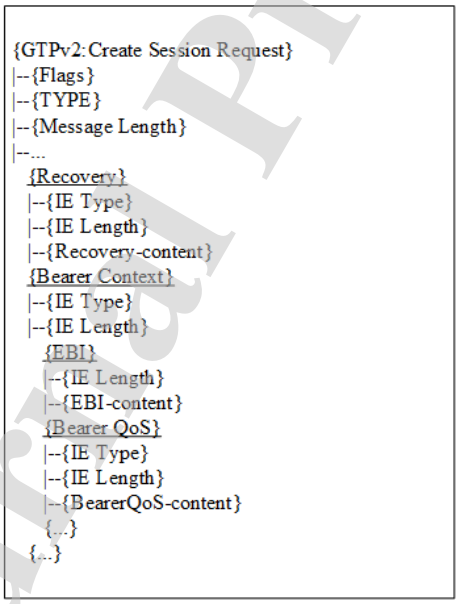

Fig. 1. Illustration of the representation of the complex frame structure. It is simplified from a packet of GTPv2.

Implementations of NB-IoT core network protocols are susceptible to bugs. The main reasons are:

- Complexities of protocols with complex frame structures. These protocols have characteristics of multiple interactions, a wide variety of information elements, and complex dependence[9]. The complexity of these protocols may easily lead to the wrong field, wrong order of states, and broken dependence of information elements in the process of implementing protocols. They are the common sources of bugs[10].
- The format specifications of the protocols are incomplete. The standards organization cannot define

every detail of the protocols clearly. Meanwhile, the communication protocols reserve some adjusted data fields to allow implementations to realize some particular functions in general. Hence implementing NB-IoT core protocols with no uniform and normative standard will lead to the existence of vulnerabilities.

We want to utilize the fuzzing technique to exploit the vulnerability of the implementations of protocols with complex frame structures. The fuzzing test generates random inputs and executes the tested program with these inputs to trigger the bugs of the target. Due to its simplicity, low consumption, and high efficiency, fuzzing becomes the most commonly used software vulnerability detection technique in real software[11, 12, 13]. And it has obtained a well developing by combining with several useful techniques, including neural network[14, 15], coverage guide[16], scheduling algorithms[17], taint analysis, and symbolic execution[11]. Conceptually, fuzzing is an optimization problem to find as many bugs as possible with limited computing resources[18]. To achieve this goal, fuzzers get some key information about the target program to improve the efficiency of the fuzzing process by the techniques mentioned above. These smart fuzzers can speculate that which input or data field may be more likely to yield more new bugs according to the information they get, and they will allocate more resources on exploiting this data field of communication protocols. For example, AFL[19] keeps the seed, which triggers new paths to the next loop, and Vuzzer[20] picks out magic bytes by data flow analysis to achieve a higher code coverage.
Given the complex frame structure of NB-IoT core network protocols, fuzzing these protocols is much harder than general network protocols by existing fuzzers. Some of the reasons are:

- Sending packets and passing the verification is difficult. Compared with file application fuzzing, protocol fuzzing needs a transmitter to send the mutated packets to the target server. Protocol fuzzing is more sophisticated because it needs to avoid being passed in the early input verification.

- Dependence is hard to control. Just like the network protocols, the fuzzer also solves the dependence such as length field, checksum field, and so on. Meanwhile, communication protocols generally have various information elements. Some information elements may also have information elements inside. The fuzzer must solve the dependence between information elements [21].

To counter these challenges, we propose an automatic hierarchy-aware fuzzing framework for fuzzing protocols with complex frame structures. Firstly, we must properly handle the format information. We can artificially determine the direction of fuzzing by the format information of protocols with complex frame structures, like Peach[22]. There is a contradiction between breadth and depth of finding paths. If the fuzzer exploits a path based on format information, it will reduce the resource allocation on exploration. The improper use of format information can lead to miss some of the bugs. Useful information can facilitate the process of fuzzing. We can never discard the format information all or partly. In the process of fuzzing, fuzzers should generate semi-valid inputs. These inputs can pass the early input verification and can trigger a bug of the target at last. If we generate the inputs fully by the format specifications, the inputs are valid, and they cannot trigger bugs. We need to generate inputs by violating some part of the formats. To select the beneficial part of the formats for fuzzing, we propose *Message Structure Tree*(MST). It contains *Rule* and two operations. We first collect the format specifications of tested protocols to generate the rule of *Message Structure Tree*(MST). HFuzz uses the rule to extract the seed file and divide it into hierarchies(we translate it to the designed tree structure in practice and introduce more details in Section 5).
Then we need to allocate more resources to the hierarchies which trigger more paths. HFuzz mutates some nodes of each hierarchy and then serialize the mutated tree to binary data. Finally, HFuzz encapsulates the mutated data as an input and sends it to the target program. *Scheduler* of HFuzz allocates the resource between hierarchies by the coverage feedback.
We evaluate the performance of HFuzz on GPRS Tunnelling Protocol version 2(GTPv2) in this paper and conduct experiments to demonstrate that HFuzz outperforms AFL and Peach. GTPv2, S1AP, and Diameter are the three protocols in the use of NB-IoT core network. While they are all protocols with complex frame structures, compared to the other two protocols, GTPv2 has no authentication, and its format is relatively simple. So we use our fuzzing framework for fuzzing the GTPv2 implementations: OAI and B*. AFL[19] is

a state-of-the-art evolutionary fuzzer, and both at achieving higher edge coverage and finding new bugs and Peach[22] is a famous model-based fuzzer that uses format information to enhance the efficiency of fuzzing for different protocols. Hence we choose these fuzzing tools as a contrast.
We make the follow contributions in our work:

- We propose the concept of *Message Structure Tree* for packets decomposition, mutation operation, and data serialization. *MST* combines the advantages of mutation- and generation-based fuzzing.
- We generate inputs of communication protocols hierarchically based on *MST*. Meanwhile, *Scheduler* adjusts the resource allocation of each hierarchy due to the coverage feedback for achieving higher coverage faster.
- We fuzz the implementations of the signaling *Create Session Request* of GTPv2. We take *SPGW* of OAI[23] and *SGW* of B* as experimental objects. The experimental result in Section 7 shows HFuzz is more efficient than AFL[19] and Peach[22].

The remainder of this paper is outlined as follows. In Section 2, we provide the details of GTPv2 and the background of HFuzz. In Section 3, we discuss the related work of fuzzing. In Section 4, we give an overview of the framework of HFuzz. In Section 5, we describe *Message Structure Tree*, one core of HFuzz. In Section 6, we explain the algorithm of hierarchy-aware scheduling, another core of HFuzz. In Section 7, we evaluate HFuzz, compared with AFL and Peach. Section 8 is our conclusions.

2. PRELIMINARY

2.1. NB-IoT Core Network and GTPv2

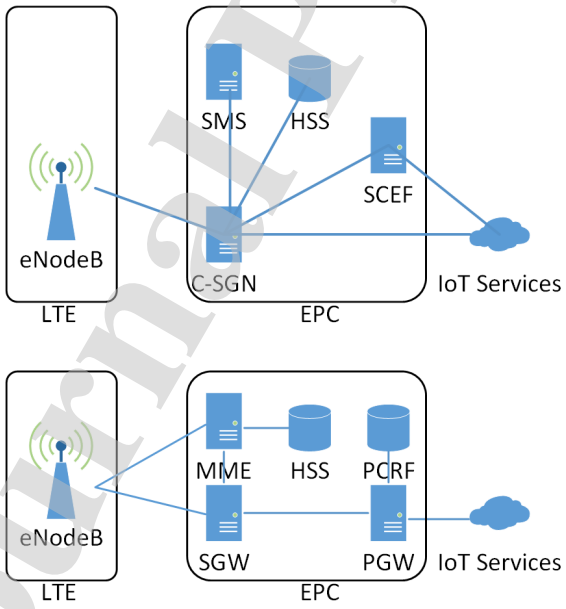

Fig. 2. Overview of NB-IoT core network with the main network elements.

As shown in Figure 2, 3GPP optimizes the architecture of the top of the figure for NB-IOT, and LTE uses the bottom architecture currently. Evolved Packet System (EPS) is standardized by the 3rd Generation Partnership Project (3GPP). It comprises Long Term Evolution (LTE) and Evolved Packet Core (EPC). NB-IoT core network adopts simplified network architecture, which optimizes the functions of NB-IoT in MME, SGW, and PGW of EPC separately to form a new network element. But, according to our research, the

manufacturers' equipments and the communities' platforms for NB-IoT do not adopt the CSGN architecture of 3GPP, but still adopt the existing EPC equipment architecture. Therefore our research also focuses on LTE architecture.

There are three main control-plane protocols in the EPC, i.e., S1AP, Diameter, and GTPv2. EPS uses GPRS Tunneling Protocol(GTP), a group of IP-based communications protocols[24], to support GPRS data transmission. GTP comprises GTP-C, GTP-U, and GTP'. GTPv2[25] is the second version of GTP-C. GTPv2 is responsible for creating, maintaining, and deleting tunnels in EPC[26]. Frequently-used types of signaling of GTPv2 are *Create Session Request*, *Modify Bearer Request*, and *Delete Session Request*. They trigger the server to send reply messages, i.e., *Create Session Response*, *Modify Bearer Response*, and *Delete Session Response*, respectively as shown in Figure 3. We choose the signaling *Create Session Request* of GTPv2 to evaluate the fuzzing framework.

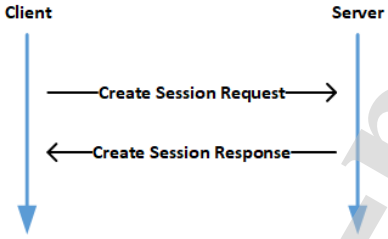

Fig. 3. Sample of signaling interaction for GTPv2.

2.2. Inputs Generation Method

There are different categories of fuzzing. Mutation-based and generation-based are the two main methods to generated inputs[27]. The mutation-based method produces inputs by mutating the selected seed files by defined strategies[19]. The generation-based method produces inputs based on the grammar of the tested program without seed files[22]. Inputs generated by mutation-based fuzzers may be rejected early in processing because the mutated data deviates too much from the expected format requirement of the target. Inputs of generation-based fuzzing may pass the early input verification of the tested program and can get deeper paths than the mutation-based method due to knowing the grammar of the target. The existing fuzzers are very difficult to use format information and keep some randomness at the same time. If we adopt the mutation-based method, some key parameters included in the seed files may be kept to produce the expected effect with some effective mutation strategies. HFuzz combines the advantages of these two methods by *Message Structure Tree* to generate semi-valid inputs.

2.3. Runtime State Information Acquisition Method

Based on the amount of known information, fuzzing techniques can be classified into three kinds: white-box, black-box, and grey-box[13]. White-box fuzzing[28, 29] has the full source code of the tested program. It knows the complete logic of the target program. black-box fuzzing mutates the real and normal inputs randomly or by predefined strategies without any information. Some black-box fuzzing tools also utilize specific information like the grammar of the target program. Grey-box fuzzing generally adopts some techniques to acquire runtime information to guide the process of fuzzing. The techniques include instrumentation[30, 31], taint analysis[32, 33], static analysis[34, 35], etc. Some fuzzing tools, such as AFL[19], Skyfire[36], and VUzzer[20], employ an evolutionary algorithm to optimize the fuzzing process. AFL uses the code instrumentation technique[31] to record the coverage of an input. If an input executes a new path, AFL will keep the input as a new seed. These inputs are used for the next round to generate inputs to achieve high coverage and an ideal outcome for vulnerabilities. Test coverage[37] is employed to measure the degree of source code which is executed in a process. The main ones of coverage criteria are function coverage, line coverage, and basic block coverage. Compared to line coverage and function

coverage, basic black coverage has fine enough granularity. The granularity of function coverage is too big to describe the details of the behavior of the program. And line coverage adopts too fine granularity, so it loses the description of characteristics of the program's behavior. HFuzz is a grey-box fuzzer that it adopts the instrument technique to record the executed basic blocks to calculate the coverage.

3. RELATED WORK

In this section, we survey the recent search of the technologies which can enhance fuzzing and explain some differences between HFuzz and some fuzzers which are related to the area of HFuzz.
Fuzzing was proposed by Miller[12] firstly in 1990. It's a highly effective technology to detect bugs in the tested program. Fuzzing tools generate masses of unexpected and incorrect test inputs. It feeds the target program these malformed data. If the target does not reject the malformed inputs, it will hang or crash. We sort out bugs from these crashes.

3.1. COVERAGE-GUIDE FUZZING

Black-box fuzzing, such as Peach[22] or Sulley[38], doesn't obtain any information about the target. Black-box fuzzing cannot optimize seed selection and mutation strategy. Coverage-guide fuzzing[19, 39, 20, 40] gets the path coverage of the target program to guide the generation of inputs in the next loop. If an input trigger a new path, the input will be added into the seed files. Coverage-guide fuzzing improves the coverage continuously in this way. Coverage-guide fuzzing is more efficient.

3.2. SCHEDULING ALGORITHM

The scheduling algorithm can apply to seed selection, mutation strategy, resource allocation, and so on. CFG[16] develops strategies to explore more low-frequency paths with the same number of inputs by using a Markov chain model. DGF[17] adopts a simulated annealing-based power schedule. DGF guides the fuzzing to the given location by allocating more resources to seeds which are closer to the target location. The scheduler of SPFuzz[21] divides into three parts: mutation scheduler, content mutation scheduler, and sequence mutation scheduler. They are responsible for scheduling different parts of protocol fuzzing. Scheduling algorithm can increase the efficiency of fuzzing but cannot directly address the problem of how to find new paths.

3.3. GRAMMAR-BASED FUZZING

Using coverage-guide technology independently is not a good way when we encounter situations where we need to generate inputs with complex structures, especially for testing protocols. The optimization based on feedback does not make the input pass the early verification more easily. Grammar-based fuzzing describes the structure of the input based on the format information, and the generated inputs can achieve a higher passrate.
Peach[22], Sulley[38], SPIKE[41], and PROTOS[42] are grammar-based fuzzers. SPIKE[41] adopts a block-based method to test the data block, which contains variables and corresponding lengths. PROTOS[42] analysis the constraint of protocols and generates test sets that violate the protocol format to discover vulnerabilities. Peach[22] is a model-based fuzzer that supports file fuzzing,and protocol fuzzing and it uses the XML file to describes the format. Sulley[38] is an open-source network protocol fuzzing tool. Sulley builds all the data chunks based on formats of the tested protocol. Now Sulley has fallen out of maintaining, and Boofuzz[43] is the successor to the Sulley. Peach and Sulley are black-box fuzzing, so they cannot adjust the seed selection, mutation strategies, and resource allocation according to the feedback.
Besides, grammar-based fuzzing also combine with other methods technologies. SNOOZE[44] is a stateful fuzzing approach to generate test cases. It uses the XML file to describe the stateful operation and the message content of the tested protocol. SPFuzz[21] proposes a three-level mutation strategy, i.e., head, content, and sequence for network protocols, and gets more paths by coverage feedback. Secfuzz[45] generates valid inputs and extras three classes, i.e., messages, payloads, and fields to use for three mutated strategies. KiF[46] is a stateful and context-aware fuzzer for SIP. Skyfire[47] proposes to process an analysis of the

format for the input in advance to improve the initial input passrate. Autofuzz[36] gets the communications between clients and servers to construct a Finite State Automaton and learn the messages syntax, which comprises the fields and types.

4. FRAMEWORK OF HFUZZ

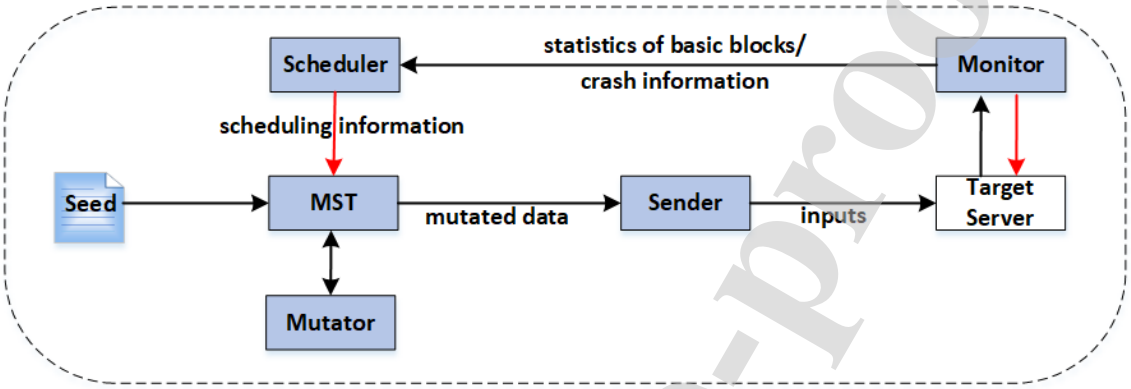

Fig. 4. A high-level overview of HFuzz.

We will introduce the components of HFuzz in this section. Figure 4 provides an overview of HFuzz. Red arrows show the flow of control information, and black arrows show the flow of data information. HFuzz comprises the following components.

4.1. Message Structure Tree

Message Structure Tree is the core component of HFuzz. *MST* contains the rule and two operations. We take *Rule* as the rule. The operations are *Decomposition* and *Serialization*. Before the start of fuzzing, *Rule* generates from the format information of the tested protocol. *MST* transfers the seed file into the defined structure by *Rule*, and *MST* accomplishes the hierarchical division of the seed file in this process. *MST* manipulates the generated tree structure to mutate the specified hierarchy of the tree structure.
At last, *MST* translates the mutated tree structure data to binary data and transmits it to *Sender*. *MST* receives the result of the resource allocation from *Scheduler* and determines the hierarchy and amount to generate mutated inputs based on the information. We will introdoce *MST* in more detail in Section 5.

4.2. Scheduler

Scheduler is another core component of HFuzz. *MST* can accomplish the hierarchy division and data mutation. *Scheduler* can recognize which hierarchy should be allocated more resources. *Scheduler* achieves resource scheduling between hierarchies based on the statistics of basic block feedback.
Scheduler executes fuzzing iteratively to realize the continuous optimization of the resource allocation. *Scheduler* takes out a fixed amount of resource and allocates resources to each hierarchy at the beginning of each loop. It allocates resources to each hierarchy averagely for the first loop, and in the next, it allocates resources according to the analysis of basic blocks in the last loop. *Monitor* transmits the record of basic blocks to *Scheduler* continually. *Scheduler* figures out the ratio of resources allocation of each hierarchy and passes the result to *MST*. The scheduling algorithms will be described in Section 6.

4.3. Mutator

Mutator mutates the input data. Our *Mutator* leverages mutation operators, such as addition, deletion, insertion, replacement of bytes, bitflip, etc. *MST* transmits the raw data to *Mutator* for mutation. *Mutator* receives data from *MST* and transmits the generated data back to *MST* after the mutation.

4.4. Monitor

Monitor acquire runtime information of the target program. *Monitor* is responsible for the following tasks. One is to monitor the running state of the target program to keep it running all the time during fuzzing. The second one is to record the crash information when some inputs crash the target program. The last one is to obtain the triggered information about basic blocks.
Monitor uses binary instrumentation to collect specific runtime information. It just instruments the main program of the target program. Every time the target program receives an input, *Monitor* records the basic blocks of it and transmits the records to *Scheduler* synchronously.

4.5. Sender

Sender is responsible for the preliminary connection to the target program. The mutated data is encapsulated and sent by *Sender*. If we generate inputs randomly based on an entire packet, there is little chance that it can pass early input verification of the target program. Even it cannot be sent out. *Sender* accomplishes the construction of underlying protocols of the tested protocol. *Sender* can increase the passrate of inputs. For example, GTPv2 is an application-layer protocol based on TCP/IP protocol, and we need only to focus on fuzzing the application-layer data regardless of the details of the transport layer, network layer, link layer, and physical layer.
Before fuzzing, we should build specific *Sender* for the target program. HFuzz is scalable for most of the protocols because we need to rebuild the *Sender* to fit different protocols.

5. MESSAGE STRUCTURE TREE

This section will explain the details of how *MST* deals with protocols with complex frame structures.

5.1. The MST Concept

Sender of HFuzz shields specific details of the underlying layer protocol. Except for the part of the underlying protocol, the frame structure of the application layer of the tested protocol is also complex. We propose the concept of *Message Structure Tree* to complete the process of hierarchy division and mutation. The *MST* contains *Rule* and two operations. The two operations, *Decomposition* and *Serialization*, run based on *Rule*. *Rule* is a novel hierarchy division method proposed to take full advantage of format information with complex frame structures. We can use these operations to accomplish hierarchy division, serialization, and some other data processing operations of the raw data. In Section 5.2, we will introduce the basic concept, operating details, and the convenient implement method of *Rule*.
Throughout this section, we use the following notation. Let R denote the set of raw binary data, which are whole application layer data of tested application-layer protocol without the data of its underlying layer protocol. Let T denote the set of *MST*, and each $t \in T$ is the *MST* representation for $r \in R$. Let C denote the set of nodes for an *MST*, and each node $c \in C$ express the content and structure of the data corresponding to this node and C_h is the set of nodes of hth hierarchy from the root hierarchy.

5.2. The Generation Method of Rule

Rule of *MST* generates from the format information of tested protocols. HFuzz uses *Rule* to translate the seed file from raw binary data to the defined tree structure. We call this tree structure *MST* representation. After some nodes of the tree structure have mutated, *MST* translates the mutated tree structure to the raw binary data as the seed file by *Rule*. *Rule* is the key to *MST*. The perfection of *Rule* depends on the amount of the format information. We can constantly enrich our known format information into *MST*. Different amounts of format information only affect the details of the generated tree structure, but the amount of known information does not make the generated *MST* representation incomplete. In this way, no matter how much the format information we know, we can make full use of it.

Rule is similar to the grammar of parsers. *MST* can divide the seed file into many hierarchies by *Rule*. In this paper, we define the hierarchy division method as that a data block can be a hierarchy, and if the data block can be divided into header and content, then the seed file can be divided into an extra hierarchy, and

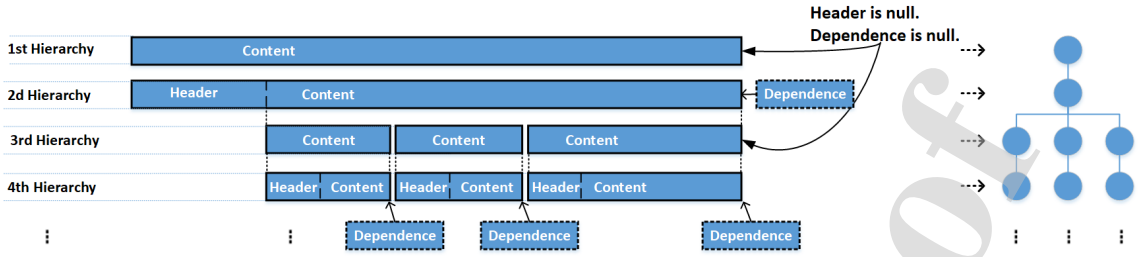

Fig. 5. An example for 4 hierarchy *MST* representation.

we define its content as the next hierarchy and so on until the data can no longer be divided. The reason we take indivisible data blocks as a hierarchy is that some seed files are from mutated inputs of which the data fields may violate the format that *MST* cannot parse them, which make sure we can use various types of seed files.

We will describe the detail of *Rule*. *MST* translates raw binary data to *MST* representation by *Rule*. The way we generate the rule is that in the range of the most format information we know, we assume that we only know a part of all the information. We gradually expand the assumed known amount of the format information from null to full in a user-defined way. Increment of the format information within a change corresponds to a new hierarchy.

As shown in Figure 5, it's an example of 4 hierarchy *MST* representation of $r \in R$. *MST* representation is an ordered tree. We define that each node $c \in C$ of the tree has three parameters, *Header*, *Content*, and *Dependence*. The *Header* defines as the data field from the head of the hierarchy of data up to the first nested element. It may comprise length, message type, flags, or some other special bytes. The *Content* is the rest of the *Header* of each hierarchy. *Dependence* describes the dependence between *Header* and *Content* of the information element. *Header* and *Content* are extracted from the seed file. *Dependence* is extracted from the format information of the tested protocol, so we circle them with dotted lines to distinguish it from the other two.

We will describe the process of generating *Rule* based on one of the increase modes by combining an example of *MST* representations, as shown in Figure 5:

- *Step 1*. Initially, we assume we know nothing about the tested protocol, so we take the seed file as an indivisible whole. Hence we can only get a one-hierarchy *MST* as the first hierarchy. This hierarchy only has one node, and its *Header* and *Dependence* are null. We take the format information which can parse the normal seed file into the above structure as the first part of *Rule*.
- *Step 2*. Increase the amount of known information, and if this information can describe the header, content, and dependence of the information elements in the hierarchy generated by the previous step. This format information can help us to construct a new hierarchy of the *MST*, like the second hierarchy. We take the increment of the format information, which can parse the seed file out to a new hierarchy as the next part of *Rule*. If we don't know more format information, the generation of *Rule* finishes.
- *Step 3*. Increase the amount of known information, and if the information can help divide the *Contend* of the hierarchy generated in *Step 2* into separate information elements, we can construct a new hierarchy of *MST*, like the third hierarchy. Like the first hierarchy, the *Header* and *Dependence* are null in this hierarchy. We take the increment of the format information as the next part of *Rule*. If we don't know more format information, the generation of *Rule* finishes.
- *Step 4*. We execute *Step 2* and *Step 3* iteratively until the assumed known information is up to the total format information we know.

When leaf nodes of $t \in T$ refer to the data which cannot be parsed, the generation of *Rule* is completed.

How should we implement this method? The reason why the above method looks complicated is because

of that it generates *Rule* from scratch. Nevertheless, when we implement this method, in reality, it is not so difficult. For example, in this paper, to test GTPv2, we find an open-source code for the implementation of GTPv2 protocol stack, which can parse the data of GTP's application layer. We apply this method into practice only by annotating the information of hierarchy based on the original program.
In the following subsections, we will detailedly describe how the operations of *MST* works by *Rule* with practical examples.

5.3. Decomposition

Through these two operations, *Decomposition* and *Serialization*, it is convenient to implement the process from extracting seed files to transmitting *Sender* the generated inputs.
There explains the detailed process of *Decomposition*. The seed file is raw binary data before we mutate it. We should translate the raw binary data $r \in R$ into the *MST* representation $t \in T$. This process of translation is called *Decomposition*. In reality, some seed files may be invalid, so *MST* cannot parse them to *MST* representation fully. Hence the degree of *Decomposition* also depends on the compliance of seed files. We initially take a seed $r \in R$ which is raw binary data as the root node of t. If the format of raw data is abnormal, where "abnormal" is defined as that the data of seed files cannot parse into *Header* and *Content* or separate information elements by *Rule*, we regard it as an indivisible data block. Its *Header* and *Dependence* are null.
Decomposition parses the raw binary data to the *MST* representations by *Rule* as follow:

- *Step 1*. *Decomposition* takes the raw binary data as the root node of the tree by *Rule*. Its *Header* and *Dependence* are null. If the raw binary data cannot be parsed by *Rule* next, the *MST* representation of this seed only has one hierarchy. If *Rule* can parse the seed, execute the next step.
- *Step 2*. Generate the child node of the node generated in the previous step. *Header*, and *Dependence* of the node generated in the last step are null. In this step, we parse the *Content* of the node of the previous step into *Header* and *Content* and save the dependence between them as *Dependence* by *Rule*. The *Header*, *Content* and *Dependence* generated just now are the parameters of the child node. And execute the next step.
- *Step 3*. If the *Content* of the node generated in the last step cannot parse into separate information elements by *Rule*, we regard it as a leaf node of the t. And if it can be parsed, we take the nested information elements as *Content* of its child nodes. *Header* and *Dependence* of these child nodes are null. And execute the next step.
- *Step 4*. Perform *Step 2* and *Step 3* on the leaf nodes iteratively until *Rule* cannot parse all the leaf nodes.

We end up with translating the raw binary data representation $r \in R$ to *MST* representation $t \in T$ shown on the right in Figure 5. Based on understanding of the format information, it can generate different *Rule*. The method of *MST* suits to any protocols, especially communication protocols with complex frame structures.

5.4. Serialization

After generating the *MST* representation t for the seed, we will manipulate the t to mutate some nodes $c \in C$ under the guidance of *Scheduler* and then translate *MST* representation $t \in T$ to raw binary data representation $r \in R$. Before *Serialization*, we need to accomplish the mutation. *MST* transmits the *Content* of the selected node of the hierarchy to mutate and then replaces the *Content* of the selected node by the mutated data. *Serialization* is responsible for serializing the mutated tree into raw binary data representation $r \in R$.
Serialization is the reverse process of *Decomposition*. *Serialization* iteratively takes depth-first order to serialize *Header* and *Content* of the nodes into binary strings. And if their *Dependence* is not null, *Serialization* should amend the *Header* of the selected nodes to meet the *Dependence* with *Content* for each node and hierarchy. *Serialization* concatenates all strings of the same father nodes to replace the *Content* of their

father node until the root node is serialized into binary data. *MST* ultimately transmits this binary string to *Sender*.
Based on the concept of *MST*, we can accomplish the mutation of the seed files hierarchically under the guidance of *Scheduler*.

6. HIERARCHY-AWARE SCHEDULING ALGORITHM

Scheduler takes charge of resource scheduling. *Scheduler* analyses the coverage of inputs which are mutated at different hierarchies to determine the allocation of resource to different hierarchy. In Section 5, we utilize the format information to generate *Rule* of *MST*. We take *MST* to realize the hierarchy division and the mutation of the appointed hierarchy of the seed file. Different information elements of hierarchies should be processed by different parts of the basic blocks of the target program. Based on the coverage feedback, we design a scheduling algorithm to allocate more resources to the hierarchy of the seed file, which is more likely to trigger bugs rather than dumb fuzz[48].

6.1. Calculation Method of Allocation

The coverage feedback technique[49] uses instrumentation[50] to gain path coverage information about the running program. We adopt *Basic Block Coverage* to evaluate the quality of the input. *Basic Block* is a sequence of assembly codes without a jump. To distinguish from the concept of *Basic Block Coverage*, we define *Basic Block Statistics* as the set of the triggered basic blocks. If new basic blocks are triggered, we regard the current hierarchy as a positive hierarchy. And we will allocate more resources to it in the next loop. With this coverage feedback technique, HFuzz can improve the coverage much faster.
To explain the method of measuring *Basic Block Statistics*, we use the following notations. Let H define the set of the order number of hierarchies, and each $h \in H$ refers to the hierarchy of *MST* representation $t \in T$. Let $|H|$ stand for the number of hierarchies of *MST*. Let *Basic Block Statistics* $BB(h)$ define the set of basic blocks, triggered by the inputs which are mutated at hierarchy h. Let N define the total resource for one loop of the fuzzing, and n_h define the resource allocated to the hierarchy h of the $t \in T$. And let $R(h)$ define the ratio of the resource allocation. $R(h)$ is the number of basic blocks $BB(h)$ divided by the sum of basic blocks of all the hierarchies:

$$R(h) = \frac{|BB(h)|}{\sum_{h \in H} |BB(i)|} \times 100\% \tag{1}$$

At the beginning of the first loop, we set the ratios of each hierarchy equal, as shown in Equation(2).

$$R(h) = \frac{1}{|H|} \times 100\% \tag{2}$$

To allocate the resource to each hierarchy of the $t \in T$ appropriately, we divide the whole fuzzing process into loops. At the beginning of each loop, n_h defines the resource allocated to the hierarchy h of the $t \in T$ to this loop.

$$n_h = R(h) \times N \tag{3}$$

Thus the fuzzer can keep hierarchy-aware of the tested program based on the coverage feedback. And *Scheduler* can allocate the resource much more accurately to each hierarchy.
To evaluate the inputs better, we propose an advanced algorithm to optimize the calculation methods for *Basic Block Statistics*.

6.2. Enhanced Algorithm for Statistics

Counting the amount of significative basic blocks is better than counting the amount of all basic blocks. During the fuzzing, some basic blocks are always executed when the target program starts, and some other basic blocks may be triggered by any inputs. Eliminating these basic blocks will aggrandize the accuracy of the description of the inputs. We think these basic blocks are meaningless for coverage calculation. We calculate the basic blocks which are triggered in the process of any inputs and denote $BB_{meaningless}$ as the set

of these basic blocks. Deprioritizing such basic blocks will result in shortening the time to the maximum of coverage.

We collect some valid inputs of the target program in normal running status, denoted as I_{valid}. I_{valid} is the set of valid inputs. $BB_{valid}(i_{valid})$ denotes the set of basic blocks triggered by a valid input $i_{valid} \in I_{valid}$. Let

$$BB_{meaningless} = \{b : \forall i_{valid} \in I_{valid}, b \in BB_{valid}(i_{valid})\}$$

, and we remove the basic blocks of BB_h which belong to $BB_{meaningless}$ simultaneously. Namely, we calculate the complement $BB_h^{'}$ of BB_h to $BB_{invalid}$ as follow:

$$BB(h)^{'} = \{b : b \in BB(h) \& b \notin BB_{meaningless}\}$$

. After suppressing the interference of the meaningless basic blocks, $BB(h)^{'}$ can provide a better representation of *Basic Block Statistics*. We use this algorithm of basic blocks calculation to improve the accuracy of ratios of the resource allocation, as shown in Equation(4).

$$R(h)^{'} = \frac{\left|BB(h)^{'}\right|}{\sum_{h \in H} |BB(h)^{'}|} \times 100\% \tag{4}$$

During the process of fuzzing, the majority of generated inputs are invalid. These inputs will be discarded early by error handling modules of the target program. Even though they do improve the space of *Basic Block Statistics*, these blocks don't make sense. These basic blocks which are responsible for error handling should be removed from this set. We generate some inputs completely randomly, denoted as $I_{invalid}$. Nearly all of these inputs are rejected by the early verification of the target program. $BB_{invalid}(i_{invalid})$ defines the set of basic blocks which are triggered by the input $i_{invalid} \in I_{invalid}$. We define BB_{error} as the set of basic blocks which exist in each member of $BB_{invalid}$ at the same time. More formally, let

$$BB_{error} = \{b : \forall i_{invalid} \in I_{invalid}, b \in BB_{invalid}(i_{invalid})\}$$

, and we remove the basic blocks of $BB_h^{'}$ which belong to $BB_{invalid}$. $BB_h^{''}$ is the complement of $BB_h^{'}$ to BB_{error}. Calculate it as follow:

$$BB(h)^{''} = \{b : b \in BB(h)^{'} \& b \notin BB_{error}\}$$

. The final ratios of the resource allocation are calculated, as shown in Equation(5).

$$R(h)^{'} = \frac{\left|BB(h)^{'}\right|}{\sum_{h \in H} |BB(h)^{'}|} \times 100\% \tag{5}$$

We choose the algorithm represented by $R(h)^{'}$ as the complete enhanced algorithm.

7. EVALUATION

Table 1. The number of basic blocks produced by AFL, Peach, HFuzz and HFuzz with enhanced algorithm on *SPGW* of *OAI* and *SGW* of *B**.

	AFL	Peach	HFuzz	Enhanced HFuzz
SPGW of OAI	868	834	941	955
PGW of B*	2361	2207	2369	2415

To determine the effectiveness of HFuzz, we evaluate in this section. We use the basic block coverage of the target program as the main metric to evaluate the performance of fuzzing tools. The goal of our evaluation is to prove two things: HFuzz gets the higher basic block coverage than AFL, Peach, and HFuzz with an enhanced algorithm can achieve the maximum coverage faster than it without.

7.1. EXPERIMENT SETUP

Most of the simulation platforms do not achieve the full functionality of EPC, so some do not implement the protocol stack of GTPv2. And given the extravagant price of commercial devices of EPC, we choose an open-source simulation platform *OAI* and a set of 4G commercial devices of *B**. We use *SPGW* of *OAI* and *SGW* of *B** to experiment. In the process of the experiment, the tested programs are both considered servers to receive the inputs. HFuzz disguises as the MME of EPC to sends the mutated inputs of GTPv2 to *SGW* or *SPGW* through S11 interface.

The experiment of *OAI* runs on a computer with 16 gigabytes RAM and 64-bit Ubuntu 16.04 LTS. The experiment of *B** runs on a server with 16 gigabytes RAM and 64-bit Centos 7.2. These tested programs have operating system limitations. They can only deploy on the specified systems. Due to these constraints, we deploy HFuzz on the same system of the tested program. Because different operating systems perform differently, we set a unified standard to measure the efficiency of different fuzzing tools on different systems. We take the number of inputs as a resource. To get the basic block coverage of the tested program, which is tested by Peach and AFL, we use *Sender* to receive the generated inputs from them and record the coverage of them by *Monitor* of HFuzz. Application-layer of the *Create Session Request* packet of GTPv2 is extracted as the seed file.

We use three main standard metrics to measure fuzzing effectiveness:

Basic Block coverage. Static analysis is applied to the binary file by *radare2*[51] before fuzzing to measure the basic block coverage. We record the total number of basic blocks. Meanwhile, the start address and end address of each basic block are also recorded. Combined with the tool we developed based on Pin[31], we can identify the basic blocks which are executed by the tested program.

Distribution of the time. We record the time when each basic block is triggered, and if the time distribution is in an earlier time of a fuzzer, that means this method is more efficient.

Bugs. Each test execution is executed under *Monitor* of HFuzz. It can record the memory corruption bugs.

We evaluate the impact of the enhanced algorithm on HFuzz, and then we evaluate the better one against the performance of existing fuzzing tools, AFL and Peach. Table 1 shows the final results of the three tools. We record HFuzz with the designed enhanced algorithm as "Enhanced HFuzz".

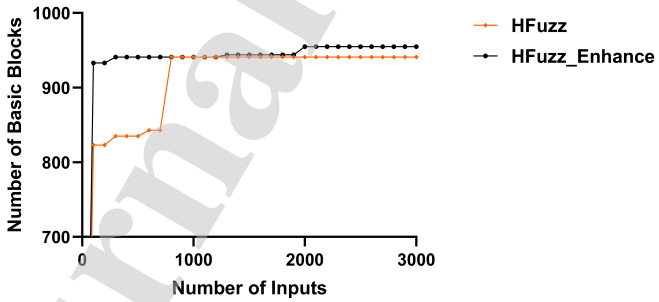

Fig. 6. The change of basic block coverage of *SPGW* of *OAI* over the number of inputs between HFuzz and enhanced HFuzz.

We verify the enhanced algorithm on *SPGW* of *OAI* for almost 6 hours(more than 3000 inputs). And we verify it on *SGW* of *B** for almost 12 hours(more than 60000 inputs). Coverage for *SPGW*, as shown in Figure 6, increases to around 950 basic blocks with the enhanced algorithm in the first 200 inputs. However, the number of basic blocks triggered by HFuzz without the enhanced algorithm reaches at around 950 in the first 800 inputs. In the end, enhanced HFuzz reaches at 955 basic blocks, and HFuzz without enhanced algorithm reaches at 941 basic blocks. The enhanced algorithm can help HFuzz to reduce the time to reach the maximum coverage.

As shown in Figure 7, in the experiment of *SGW* of *B**, HFuzz with the enhanced algorithm triggers around 2350 in the first 4000 inputs, HFuzz without the enhanced algorithm reaches at around the same

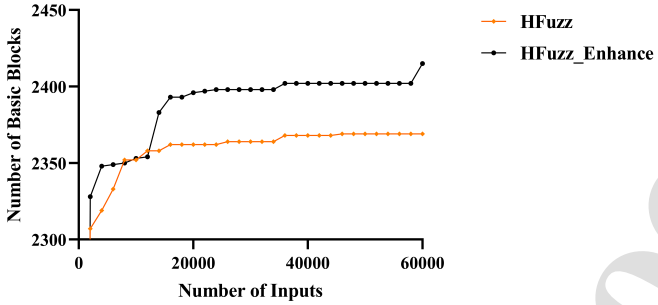

Fig. 7. The change of basic block coverage of *SGW* of *B** over the number of inputs between HFuzz and enhanced HFuzz.

amount basic blocks almost in the first 7800 inputs. In the end, enhanced HFuzz achieves 2415 basic blocks and HFuzz without the enhanced algorithm only gets 2369 basic blocks. Combined with the experimental data analysis of *SGW* and *SPGW* above, the enhanced algorithm can accelerate the process of reaching coverage stability and may help HFuzz achieve higher coverage at the same time.

7.2. EVALUATING WITH AFL AND PEACH

In this experiment, we compare the enhanced HFuzz with AFL and Peach. As shown in Figure 8, it is

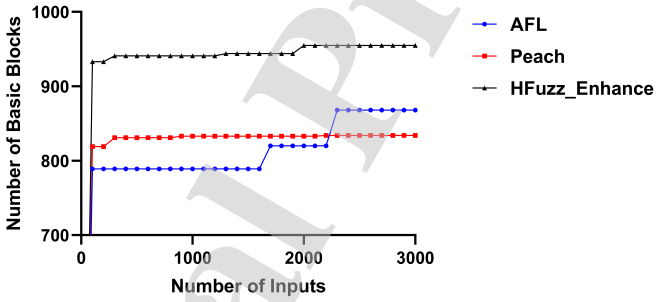

Fig. 8. The change of basic block coverage of *SPGW* of *OAI* over the number of inputs among enhanced HFuzz, AFL, and Peach.

the experiment of *SPGW* in *OAI*. Enhanced HFuzz triggers 955 basic blocks at termination while AFL and Peach achieve 868 and 834 basic blocks, respectively.

As shown in Figure 9, it is the experiment of *SGW* in *B**. Coverage for *SGW* increases to 2415 basic blocks with Enhanced HFuzz while AFL and Peach rise to 2361 and 2207, respectively.

To get a more reasonable number of increment of basic blocks which are triggered by the initial step, we randomly select 50 normal GTPv2 packets respectively from OAI and B* and executed the target programs with these inputs. Then we record the sums of the triggered basic blocks, and we denote 743 and 2086 basic blocks as the initial numbers of OAI and B*, respectively. Therefore, the increment of the first step is the number of the basic blocks in the first step minus the initial number. It is clearly in Figure 10 and Figure 11 that compared to other tools, enhanced HFuzz not only discovers the new paths early, but also the new paths mostly gather in the early stage.

In conclusion, HFuzz can acquire better basic block coverage than AFL and Peach based on *Message Structure Tree*. *MST* can amend every other node and every hierarchy to make this input as valid as possible if we fuzz a deep node of a seed file in *MST* representation. Hence the inputs can pass the early input verification to achieve better coverage. And in the follow-up process, HFuzz still achieves the highest coverage. *Scheduler* makes HFuzz exploit the hierarchy, which can trigger more basic blocks. Through the comparison of

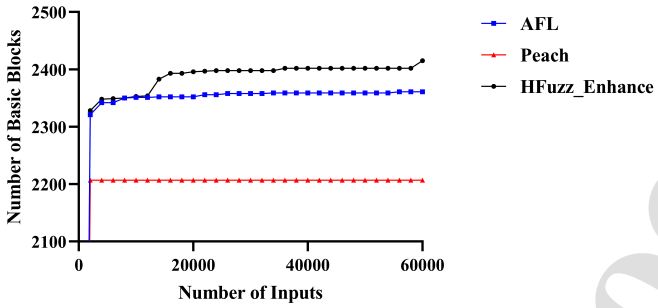

Fig. 9. The change of basic block coverage of *SGW* of *B** over the number of inputs among enhanced HFuzz, AFL, and Peach.

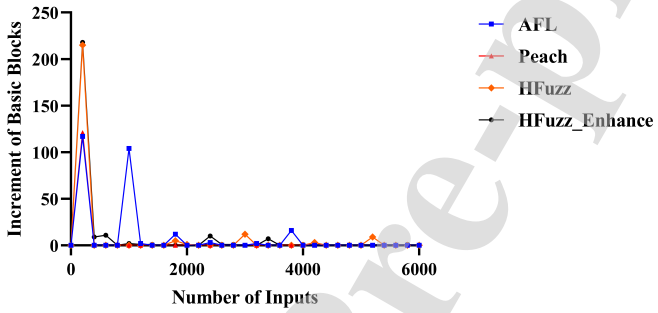

Fig. 10. The distribution of the amount of triggered basic blocks of B*. The X-axis denotes the number of inputs. Y-axis denotes the unique number of basic blocks found.

experimental data, the enhanced algorithm shortened the time to exploit an equal number of basic blocks and get better coverage than AFL and Peach.

7.3. Bug Analysis

During the testing process, HFuzz finds a new bug. It is a heap overflow bug of SPGW in OAI. After analyzing the log, this heap error is generated from the function "nwGtpv2cProcessUdpReq()" of file "NwGtpv2ParseInfo.c". If a malformed data packet that contains an excess amount of the same type of information elements that exceed the upper limit, the crash would be triggered.

8. CONCLUSION

To test NB-IoT core network protocols, we propose an automatic fuzzing framework optimizing the fuzzing process by the novel hierarchy-aware and scheduling algorithm. Compared to general network protocols, protocols of NB-IoT core network in use, such as S1AP, GTPv2, and Diameter, all have complex frame structures. We overcome the challenges of fuzzing protocol, especially some with complex frame structures by HFuzz. We propose a new concept *Message Structure Tree* to process the seed files and translate the seed files from raw binary type to a tree structure by the rule generated of *MST*. We define a new method to deal with the format information that we cannot confuse by choice of format information. After *MST* processes it, the original seed file is divided into several hierarchies. Then we take a hierarchy-aware scheduling algorithm to achieve the resource allocation between hierarchies. In this way, we can get a balance between exploration and exploitation. And on that basis, we propose the enhanced calculation method of coverage to eliminate distractions with unnecessary basic blocks. It shortens the time to maximize coverage. In experiments, we prove that HFuzz becomes more powerful with the enhanced algorithm and

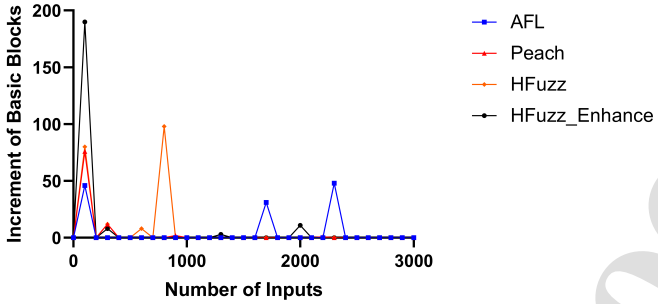

Fig. 11. The distribution of the amount of triggered basic blocks of OAI. The X-axis denotes the number of inputs. Y-axis denotes the unique number of basic blocks found.

HFuzz is more efficient than AFL and Peach in fuzzing protocols with complex frame structures from three indicators.

ACKNOWLEDGMENTS

Thanks for the support of the National Natural Science Foundation of China (No. 61872836) and the Fundamental Research Funds for the Central Universities (No. 500419810).

References

[1] R. Sanchez-Iborra, M.-D. Cano, State of the art in lp-wan solutions for industrial iot services, Sensors 16 (5) (2016) 708.
[2] Lte-m, nb-lte-m and nb-iot: Three 3gpp iot technologies to get familiar with, `https://www.link-labs.com/blog/lte-iot-technologies`, accessed,April,2019.
[3] Huawei and partners leading nb-iot standardization, `https://www.prnewswire.co.uk/news-releases/huawei-and-partners-leading-nb-iot-standardization-528516901.html`, accessed,April,2019.
[4] Narrow band iot and m2m - global narrowband iot – lte-m networks – march 2019, `https://gsacom.com/paper/global-narrowband-iot-lte-m-networks-march-2019/`, accessed,April,2019.
[5] M.-D. C. . Ramon Sanchez-Iborra, State of the art in lp-wan solutions for industrial iot services, `https://www.gsma.com/iot/extended-coverage-gsm-internet-of-things-ec-gsm-iot/`, accessed,April,2019.
[6] R. Sanchez-Iborra, J. Sanchez-Gomez, J. Ballesta-Viñas, M.-D. Cano, A. Skarmeta, Performance evaluation of lora considering scenario conditions, Sensors 18 (3) (2018) 772.
[7] H. Suo, J. Wan, C. Zou, J. Liu, Security in the internet of things: a review, in: 2012 international conference on computer science and electronics engineering, Vol. 3, IEEE, 2012, pp. 648–651.
[8] Y. Xu, J. Ren, G. Wang, C. Zhang, J. Yang, Y. Zhang, A blockchain-based nonrepudiation network computing service scheme for industrial iot, IEEE Transactions on Industrial Informatics 15 (6) (2019) 3632–3641. doi:10.1109/TII.2019.2897133.
[9] L. J. Rodríguez-Aragón, Tema 4: Internet y teleinformática, Recuperado desde: http://www.uclm.es/profesorado/licesio/Docencia/IB/IBTema4. pdf.
[10] A. Walz, A. Sikora, Exploiting dissent: towards fuzzing-based differential black box testing of tls implementations, IEEE Transactions on Dependable and Secure Computing.
[11] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, W. Liu, A systematic review of fuzzing techniques, Computers & Security 75 (2018) 118–137.
[12] B. P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of unix utilities, Communications of the Acm 33 (12) (1990) 32–44.
[13] R. McNally, K. Yiu, D. Grove, D. Gerhardy, Fuzzing: the state of the art, Tech. rep., DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA) (2012).
[14] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, S. Jana, Neuzz: Efficient fuzzing with neural program learning, arXiv preprint arXiv:1807.05620.
[15] X. Yan, B. Cui, Y. Xu, P. Shi, Z. Wang, A method of information protection for collaborative deep learning under gan model attack, IEEE/ACM Transactions on Computational Biology and Bioinformatics PP (2019) 1–1. doi:10.1109/TCBB.2019.2940583.
[16] M. Böhme, V.-T. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as markov chain, IEEE Transactions on Software Engineering.
[17] M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury, Directed greybox fuzzing, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 2329–2344.

[18] M. Woo, S. K. Cha, S. Gottlieb, D. Brumley, Scheduling black-box mutational fuzzing, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, ACM, New York, NY, USA, 2013, pp. 511–522. doi:10.1145/2508859.2516736.
URL http://doi.acm.org/10.1145/2508859.2516736

[19] M. Zalewski, American fuzzy lop (afl), December.

[20] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, H. Bos, Vuzzer: Application-aware evolutionary fuzzing., in: NDSS, Vol. 17, 2017, pp. 1–14.

[21] C. Song, B. Yu, X. Zhou, Q. Yang, Spfuzz: A hierarchical scheduling framework for stateful network protocol fuzzing, IEEE Access 7 (2019) 18490–18499.

[22] The peach fuzzer platform, https://www.peach.tech/products/peach-fuzzer/, accessed,April,2019.

[23] Openairinterface, https://www.openairinterface.org, accessed,April,2019.

[24] Wikipedia contributors, Gprs tunnelling protocol — Wikipedia, the free encyclopedia, [Online; accessed 15-July-2019] (2019).
URL https://en.wikipedia.org/w/index.php?title=GPRS_Tunnelling_Protocol&oldid=900425452

[25] 3gpp evolved packet system (eps); evolved general packet radio service (gprs) tunnelling protocol for control plane (gtpv2-c); stage 3, https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1692, accessed,April,2019.

[26] Author: Frédéric Firmin, 3GPP MCC, The evolved packet core, https://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core, accessed,April,2019.

[27] P. Oehlert, Violating assumptions with fuzzing, IEEE Security & Privacy 3 (2) (2005) 58–62.

[28] J. DeMott, The evolving art of fuzzing, DEF CON 14.

[29] P. Godefroid, Random testing for security: blackbox vs. whitebox fuzzing, in: Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), ACM, 2007, pp. 1–1.

[30] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: ACM Sigplan notices, Vol. 42, ACM, 2007, pp. 89–100.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Acm sigplan notices, Vol. 40, ACM, 2005, pp. 190–200.

[32] I. Haller, A. Slowinska, M. Neugschwandtner, H. Bos, Dowsing for overflows: A guided fuzzer to find buffer boundary violations, in: Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13), 2013, pp. 49–64.

[33] S. Bekrar, C. Bekrar, R. Groz, L. Mounier, A taint based approach for smart fuzzing, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, 2012, pp. 818–825.

[34] J. Kinder, F. Zuleger, H. Veith, An abstract interpretation-based framework for control flow reconstruction from binaries, in: International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer, 2009, pp. 214–228.

[35] S. Sparks, S. Embleton, R. Cunningham, C. Zou, Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting, in: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), IEEE, 2007, pp. 477–486.

[36] J. Wang, B. Chen, L. Wei, Y. Liu, Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017, pp. 579–594.

[37] Wikipedia contributors, Code coverage — Wikipedia, the free encyclopedia, [Online; accessed 8-July-2019] (2019).
URL https://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=892598579

[38] Sulley fuzzer, https://github.com/OpenRCE/sulley, accessed,April,2019.

[39] syzkaller, https://github.com/google/syzkaller, accessed,April,2019.

[40] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution., in: NDSS, Vol. 16, 2016, pp. 1–16.

[41] An introduction to fuzzing: Using fuzzers (spike) to find vulnerabilities, https://resources.infosecinstitute.com/intro-to-fuzzing/#gref, accessed,April,2019.

[42] J. Röning, M. Lasko, A. Takanen, R. Kaksonen, Protos-systematic approach to eliminate software vulnerabilities, Invited presentation at Microsoft Research.

[43] Boofuzz, https://boofuzz.readthedocs.io, accessed,April,2019.

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

[45] P. Tsankov, M. T. Dashti, D. Basin, Secfuzz: Fuzz-testing security protocols, in: Proceedings of the 7th International Workshop on Automation of Software Test, IEEE Press, 2012, pp. 1–7.

[46] H. J. Abdelnur, O. Festor, et al., Kif: a stateful sip fuzzer, in: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications, ACM, 2007, pp. 47–56.

[47] S. Gorbunov, A. Rosenbloom, Autofuzz: Automated network protocol fuzzing framework, IJCSNS 10 (8) (2010) 239.

[48] C. Miller, How smart is intelligent fuzzing-or-how stupid is dumb fuzzing, Independent Security Evaluators.

[49] P. Tsankov, M. T. Dashti, D. Basin, Semi-valid input coverage for fuzz testing, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, 2013, pp. 56–66.

[50] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, J. Chau, Framework for instruction-level tracing and analysis of program executions, in: Proceedings of the 2nd international conference on Virtual execution environments, ACM, 2006, pp. 154–163.

[51] radare2, https://www.radare.org/r/, accessed,April,2019.

Highlights：

1. Automatic hierarchy-aware scheduling fuzzing framework is suitable for NB-IoT core network protocols
2. We use **Message Structure Tree** to translate the seed file into the defined tree structure, and mutating its nodes to can effectively improve the passrate and code coverage.
3. After completing hierarchy division, allocating resources among hierarchies according to feedback by the scheduling algorithm can improve the code coverage faster.
4. Eliminating some basic blocks of interference can effectively improve the efficiency of fuzzing.
5. HFuzz outperforms AFL and Peach on coverage and it finds a new bug of OAI.

Declaration of interests

The authors declare that they do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

XINYAO LIU received the BS degree from Beijing University of Posts and Telecommunications, China, in 2016. He is currently working toward the Ph.D. degree with Beijing University of Posts and Telecommunications, China. His research interests include security and privacy issues in distributed wireless networks, cloud computing and Internet of Things.

BAOJIANG CUI received the BS degree from the Hebei University of Technology, China, 1994, the MS degree from the Harbin Institute of Technology, China, in 1998, and the Ph.D. degree in control theory and control engineering from Naikai University, China, in 2004. He is currently a professor at the School of Computer Science, Beijing University of Posts and Telecommunications, China. His main research interests include detection of software, cloud computing and the Internet of Things.

JUNSONG FU received his Ph.D. degree in communication and information system from Beijing Jiaotong University in 2018. He is now serving as an assistant professor in the school of cyberspace security in Being University of Posts and Telecommunications. His research interests include in-network data processing, network security and information privacy issues in distributed systems and Internet of Things.

JINXIN MA received his Ph.D. degree in cyberspace security from Beihang University, Beijing, China in 2014. Currently, he is working as a research scientist in CNITSEC, Beijing, China. Meanwhile, he is also a research supervisor in Beijing University of Posts and Telecommunications, Beijing, China. His research interest includes software and network security and program analysis.