eFuzz: A Fuzzer for DLMS/COSEM Electricity Meters

Henrique Dantas, Zekeriya Erkin, and Christian Doerr
Cyber Security Group, Depart. of Intelligent Systems
Delft University of Technology
Mekelweg 4, 2628 CD, Delft, Netherlands
hndantas@gmail.com,{z.erkin, c.doerr}@tudelft.nl

Raymond Hallie
European Network for Cyber Security (ENCS)
The Hague, Netherlands
raymond.hallie@encs.eu

Gerrit van der Bij
Riscure BV
Delft, Netherlands
vanderbij@riscure.com

ABSTRACT

Smart grids enable new functionalities like remote and micro management and consequently, provide increased efficiency, easy management and effectiveness of the entire power grid infrastructure. In order to achieve this, smart meters are attached to the communication network, collecting fine granular data. Unfortunately, as the smart meters are limited devices connected to the network and running software, they also make the whole smart grid more vulnerable than the traditional grids in term of software problems and even possible cyber attacks. In this paper, we work towards an increased software security of smart metering devices and propose a fuzzing framework, eFuzz, built on the generic fuzzing framework Peach to detect software problems. eFuzz tests smart metering devices based on the communication protocol DLMS/COSEM, the standard protocol used in Europe, for possible faults. Our experiments prove the effectiveness of using an automated fuzzing framework compared to resource demanding, human made software protocol inspections. As an example, eFuzz detected between 10 and 40 bugs in different configurations in less than 3 hours while a manual inspection takes weeks. We also investigate the quality of the eFuzz results by comparing with the traditional non-automated evaluation of the same device with respect to scope and efficiency. Our analysis shows that eFuzz is a powerful tool for security inspections for smart meters, and embedded systems in general.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

SEGS'14, November 7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2667190.2667194.

Keywords

Automated Testing; Fuzz Testing; Software Vulnerabilities

General Terms

Security

1. INTRODUCTION

Critical infrastructures are essential assets to the correct functioning of the economy and society. One of these infrastructures is the energy grid, which is one of the most fundamental and visible commodities as any disruption in its supply can have profound effects on virtually everything else. Currently, the energy grid is undergoing significant transformation in the direction of smart grids that will enable a new range of applications, including but not limited to fine grained tariffs to decrease consumption fluxes, remote access to metering data and better management of micro-generation.

The evolution in the energy sector has undoubtedly potential to improve the way society makes use of electricity, however, it also opens the door to a new class of threats. As the current devices gain networking capabilities and the ability to execute remote commands they also become more exposed and more attractive to adversaries. Therefore, it is important to perform security audits of the different grid components.

Particularly, networked controllable electricity meters, typically referred to as smart meters, are a fundamental part of the smart grid. Smart meters are situated in each household, hence are broadly available and can be easily probed by many people. As the physical protection is limited, it is safe to assume that curious customers will inspect the devices to try and understand how they work. Furthermore, its communication capabilities would also enable an adversary of gaining control remotely. If a vulnerability is found that, for example, allows tampering with the reported usage, it can have important repercussions. The impact will be significant if the vulnerabilities are published online, thus endangering a large installation base of meters to be exploited. Since the smart meters are relatively new and there is little standardization with respect to security, many different ven-

dors develop their own software, and we envision that smart meter software is vulnerable to attacks.

Some of the plausible attack scenarios are as follows.

- *Disclosure of sensitive data* — Vulnerabilities in the smart meter that result in the leakage of consumption information can be used to ascertain if the home or shop owners are away.
- *Meter tampering* — Attackers may find desirable to inflate the utility bill of victims or unlawfully reduce their own.
- *Pivoting* — Penetration of meter defenses can be used for reconnaissance purposes or exploitation of other parts of the system. If the meter is connected to a domestic network the hacker could use the meter to penetrate other devices in the house. Or, more creatively, configure it to make free calls or grant Internet connectivity via the GSM connection that is common in smart meters. Alternatively it can reveal valuable information about sensitive parts of the smart grid network.
- *Remote command execution* — If an adversary gains complete access to a meter, it may be used to extort individual victims, similarly to ransomware. Meters could be switched off, or internal security mechanisms be silently disabled so the electric installation or the meter itself would be damaged.
- *Grid destabilization* — Given the remote accessibility of smart meters, an attacker could gain control of hundreds or thousands of identical metering devices and exploit vulnerabilities to simultaneously switch heavy loads such as electric cars on and off, with the intent to destabilize and bring down the power grid.

One technique that is widely used to find vulnerabilities in proprietary software is fuzzing. Very succinctly, it consists of trying numerous combinations of inputs, violating protocol or file format rules, to see how the target responds to unexpected requests. One of the simplest way to perform a fuzzing attack is to manually select the input. First one develops a crude version of a protocol client and then chooses a set of values the tester believes may cause software faults in the target system. However, this approach has various limitations: it is hard to fuzz complex protocols, reuse of code is not always possible, and the developers have to focus on more than the target protocol. To cope with such limitations, fuzzing frameworks have been developed, such as SPIKE [1], SNOOZE [2] and Peach [7].

These fuzzing frameworks typically target software applications running on traditional computing platforms, such as desktops. To monitor the working state of the targeted software, they might for example attach debuggers to the process or inspect memory consumption. Unfortunately, in embedded systems like smart meters, these capabilities are not always available and the fuzzing framework is usually physically separated from the investigated target. Therefore, to fuzz an embedded system like a smart meter, it is necessary to use a framework that supports the communication protocols available in the target and that allows specific types of monitoring developed for limited devices.

Fuzzing provides several advantages over traditional security evaluations such as manual source code reviews, which motivate proceeding with this approach. A fuzzer is an automated technique which means it can be run inexpensively over long periods of time. Thus, it can provide a level of coverage hard to replicate by humans. An additional benefit is that it does not require access to the source code running on the target. Consequently, a higher number of parties can investigate the security of, in this case, smart meters without requiring formal agreements with the vendor. Despite its advantages there are virtually no fuzzers for DLMS/COSEM meters.

In this paper, we address this void and present a new fuzzing framework, developed particularly for smart meters. The main purpose of eFuzz is to investigate vulnerabilities in smart meter communication protocols. In the European market, DLMS/COSEM is the *de facto* communication protocol for smart metering application, hence we present this protocol as a use case in this paper to conduct our tests.

We provide an abstract, customizable and versatile fuzzing framework for smart meters that is not restricted to any specific device. When evaluating the functionality and effectiveness of our framework, we present the results of a vulnerability test performed on a smart meter currently deployed in the Netherlands. The tests conducted on this specific smart meter are also compared with traditional approach, analysis made by a human inspector, and the results are given.

The outline of this paper is as follows: Section 2 highlights relevant related work. Section 3 introduces fundamentals of fuzzing and the DLMS/COSEM communication protocol. This is followed by Section 4 that addresses our proposed prototype, eFuzz. Section 5 evaluates the performance of our automated fuzzing approach and discusses the effectiveness of different fuzzing strategies. Finally, we conclude in Section 6.

2. RELATED WORK

Fuzzing was originally developed at the University of Wisconsin in 1988 [15] and as one of the first applications, command-line UNIX programs were tested [16]. The success of this technique was astonishing as this approach induced crashes in 25 to 33% of test cases. Since then, fuzzing has grown in sophistication and scope. Recent literature describes several techniques such as taint analysis that aim to increase the efficiency of fuzzers [3, 4, 20].

In essence, this technique first marks untrusted data (e.g. input forms) as tainted. Then it tracks its propagation and which variables are affected by it during runtime. This mark and track method exposes the flows between data sources and sinks. Such data progress information can be used to identify the most relevant paths of execution, which can later be used to improve the efficiency of the fuzzer. Another important component is being able to monitor faults caused by the fuzzer. However, such techniques, in the current state, cannot be readily applied in embedded systems. For example, in these systems it is not always possible to have access to the binaries running on the target.

With the prominence of the Internet of Things, there is a growing need to evaluate the security of embedded devices. These systems assume numerous shapes and sizes and support a myriad of protocols and interfaces. Particularly for fuzzing, the heterogeneity of targets significantly increases the cost of tests as fuzzers needs to be rewritten to support the specific protocols and interfaces available on the device under test. Currently the most common way to deal with

such difficulties is to use dedicated hardware to enable compatibility, both on the interface and protocol. [5, 17] feature various examples of fuzzing setups.

Currently, available literature [11, 13, 18, 9, 8, 21] on smart grid security are focused on higher level designs or guidelines that aim to raise awareness and assist stakeholders implementing this complex network in a relatively secure way since this is a nascent field.

In comparison, few researchers have contributed tools to evaluate devices already in the market. In particular for smart meters, one of the exceptions is the commercially available ProtoPredator for Smart Meters (PP4SM) [14] that enables fuzzing of ANSI C12.18 meters through the optical interface.

On the non-commercial side, Termineter [12] is an open-source python framework for security testing that supports ANSI C12.18 and C12.19, standards that are commonly used for smart metering in the United States. Unlike PP4SM, Termineter is not a fuzzer, it is instead an implementation of the protocol. Unfortunately C12.x and DLMS/COSEM are distinct standards which makes it difficult to adapt the aforementioned software applications to support DLMS/COSEM. Moreover, due to the particular syntax used to specify protocol packets in Peach, the C12.x implementation provided by Termineter cannot be adapted without significant modifications. Furthermore, comparing the advantages provided by Peach against the possibility of expanding Termineter to support DLMS/COSEM, it was concluded that the former would be a better approach.

For DLMS/COSEM, the authors are not aware of any similar applications, commercial or non-commercial, further reinforcing the need for a tool such as eFuzz.

3. BACKGROUND

This section first provides a high level overview of fuzzing frameworks, followed by a description the DLMS/COSEM protocol used in European smart meters.

3.1 Fuzzing

In general, fuzzers can be divided in three categories with respect to how they generate new packets or files. The first one is to feed completely random data as input to the target. This method is simple but is less likely to yield good results. In the second category, the mutation-based fuzzers alter pre-generated valid inputs in hope of discovering a vulnerability. In the third category, generation-based fuzzers create files or packets from scratch with the same intent based on a user specified model.

Compared to others, the main advantage of the second category is that little knowledge of the file or protocol format is required and thus the preparation time is modest. However the code coverage is dependent on the coverage provided by the original valid inputs. On the other hand, constructing new files or packets can only be done if the tester has sufficient knowledge of the file or protocol format. But this approach results in a more comprehensive test case suite and thus, it is more likely to find vulnerabilities. eFuzz belongs in this category.

To assist the creation of fuzzers various fuzzing frameworks were created. Dave Aitel proposed a block-based fuzzing framework designated SPIKE [1]. The framework was developed to simulate network protocol clients and automate black-box testing. It is implemented as a C-like API and scripting language. These are used to leverage the tester's knowledge of the protocol.

SNOOZE [2] is a network protocol fuzzer that implements a stateful fuzzing approach. Therefore, a tester can describe the stateful operation of the protocol and specify the messages to be generated in each state. Moreover, it provides fuzzing primitives that are specific to certain attacks and thus allow the tester to direct her efforts solely on a class of vulnerabilities.

Peach [7], the brainchild of Michael Eddington, has become one of the most mature and, arguably, the most widely used fuzzing framework. Hence this framework was chosen for the development of eFuzz. Peach is capable of both generation and mutation based fuzzing. To define the structure, type information and relationships in the data to be fuzzed, Peach uses manually crafted Peach Pit files. These files describe the protocol model and define how such model will be used to test the target.

The modular architecture of Peach encourages code reuse and provides easy extendability. The main advantages of using Peach instead of developing a fuzzer from the ground up are the abstraction of the mutation strategies, the simple way the protocol can be modelled and the modularity of the various components (custom or built-in).

3.2 DLMS/COSEM Communication Protocol

The Device Language Message Specification (DLMS) consists of a general concept for abstract modelling of communication entities. On top of this, the *COmpanion Specification for Energy Metering* (COSEM) provides a set of standards that define the rules for data exchange between smart grid devices, such as an energy meter and a data accumulator. Together they provide several features, such as (a) an object model to view and access the different functionalities of a meter, (b) an identification system for all data, (c) a method for communicating with the model and (d) a transport layer to accommodate the information flows between the meter and other devices. The protocol [**?**, **?**, **?**, **?**] is based on the *Open System Interconnection* (OSI) seven-layer model. However in the case of smart meters they are practically collapsed into four - physical, data link, transport and application layers.

In a similar fashion to the OSI model, the physical layer in DLMS/COSEM defines how to transfer information to and from the meter. The data layer provides the messaging methods to modify data and communicate with the device. The transport layer enables data transfer based on the IPv4 network. Finally the application layer represents the functional aspects of the energy meter so applications can access them.

Prior to exchanging metering information an association should be established, initiated by the client through the object model interface. From that moment, the server is also able to send notification without an explicit request. DLMS/ COSEM supports authentication and confidentiality services based on symmetric key encryption.

As explained above DLMS/COSEM is a connection oriented protocol and it encapsulates its traffic differently depending on the interface. Since eFuzz uses a physical connection the following explanations will focus on the layers for this particular configuration. Figure 1 features a high-level overview of DLMS/COSEM request over serial port.

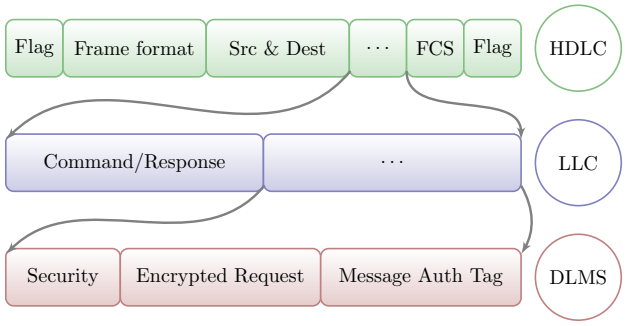


Figure 1: DLMS/COSEM packet for direct connections.

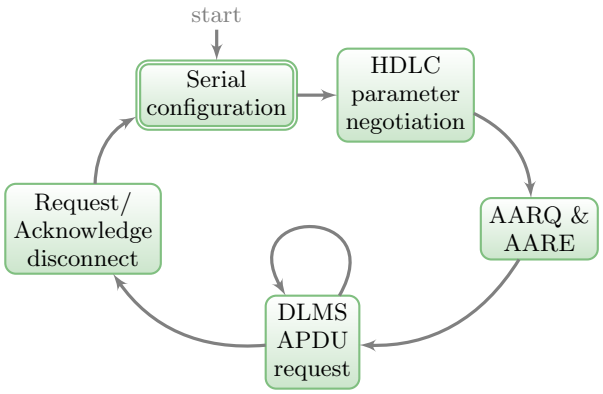


Figure 2: A complete DLMS/COSEM flow including establishing a connection, making a request, reading the response and closing the connection. This example is for the physical interface.

The outer layer is the physical layer, not shown in the figure. For the current set-up it consists of the serial communication over USB. When using a direct connection in DLMS/COSEM, the High-Level Data Link Control (HDLC) protocol is used. This data link layer, show on the top of the figure, is composed of opening and closing flags, the type of frame, source and destination identifiers, payload and checksum (Frame Check Sequence or FCS) to ensure data integrity. The Logical link control (LLC), in the middle, is the upper sub-layer of the data link layer. It is short in length and is used to specify is the payload is a response or a command. The application layer, shown on the bottom, contains the DLMS protocol data unit. Its content specifies the level of security it is using and, if applicable, proceeds to transmit the encrypted request followed by the authentication tag.

The target meter is configured to use encryption and authentication to protect the transport of data, for this purpose DLMS/COSEM uses Galois Counter Mode of operation with AES-128 (AES-GCM).

The encryption and decryption operations use the concatenation of the system title and frame counter as initialization vector. The former is device specific and is announced in the configuration phases of the session. While the latter is transmitted alongside the ciphertext. The key for AES-GCM is specific for each device, in other words, both the meter and the PC have their own. Lastly, the authentication data includes the symmetric authentication key. The decryption process is similar but adds a verification step for the authentication tag.

It is also insightful to have an understanding of the complete communication flow that a simple request entails. A generic example over physical connection is show in Figure 2. Serial connection is divided into three distinct phases. First, a handshake takes place where the meter identifies itself and announces the serial configuration it supports, including baud-rate and other serial related parameters. Second, the client (PC) initializes the HDLC (High-Level Data Link Control, a data link layer protocol) link by transmitting a frame to the meter. After reception the meter replies with the frame configurations it supports. The third and last step to complete the connection establishment is the pair AARQ/AARE (Application Association Request and Response, respectively). The former is sent by the PC and the latter is the response of the meter. This exchange is used to establish more parameters specific to the DLMS layer (i.e. independent of the outer-layers which depend on the interface). If defined in the security configurations, these are the first packets to be encrypted and authenticated.

Finally, we are able to start the actual DLMS/COSEM communication. In this session, the client may query the server for usage data, time and date, or modify the parameters of the meter, such as increase the number of available tariffs, clear the logs, etc. Although the previous steps may change from meter to meter and from interface to interface, the DLMS *Application Layer Protocol Data Units* (APDUs) do not. This makes this layer the most interesting point of attack for tools like eFuzz as it is prevalent across compliant meters. Therefore, this makes the application very versatile as it is simple to adapt it to different devices.

4. eFuzz: A FUZZER FRAMEWORK FOR SMART METERS

As previously mentioned eFuzz is built on top of Peach (version 2.3.9) to facilitate development. However a fuzzer for DLMS/COSEM smart meter has several specificities that distinguish it from fuzzers targeting more common network protocols.

DLMS/COSEM is relatively strict with respect to user input, since it only allows access to a limited set objects present in the meter. One of the consequences is that using a completely mutation based fuzzer is not as useful since the input space accepted by the target is smaller. In comparison, Internet protocols support a much vaster input space. Due to its niche application, smart metering, there are less tools to support and assist developers working with the protocol. For example, the most used network protocol analyzer, Wireshark, does not include a dissector for DLMS/COSEM. This forces the user to map individual bytes to the specification, resulting in a steep learning curve. Furthermore, since meters have limited computational capabilities it is necessary to have an optimized implementation of the fuzzer, and be able to easily modify its lower level components when required. Finally smart meters, as well as other embedded systems, typically support interfaces that are not often used in other contexts. Writing a fuzzer to make use of such hardware requires prior understanding of such equipment.

In summary, due to the particularities of smart meters, writing a fuzzer for such a target requires a deep knowl-

edge of the protocol and its hardware than for other more widespread network devices.

Peach is divided in different components according to their task which are all orchestrated by the aforementioned Pit file. In this context, the most relevant ones are the "transformers", "fixups" and "publishers". Peach comes pre-installed with several instances of each component. However, since embedded systems are not typical targets for fuzzing, it was necessary to develop custom versions to communicate with the meter under test.

According to [6], transformers perform static modifications or encoding of the parent elements. In eFuzz, a custom transformer was written to encrypt the APDUs. Fixups are similar to transformers but instead operate on data retrieved from an another element other than their parent. When generating DLMS/COSEM packets, a fixup is used to determine the frame check sequence (FCS). A reference implementation in C can be found in [19]. Finally, publishers are responsible for the I/O communication, in this case serial. eFuzz uses the PySerial [10] library to enable cross-platform serial communication. Figure 3 illustrates the aforementioned components as well as their interactions and dependencies.

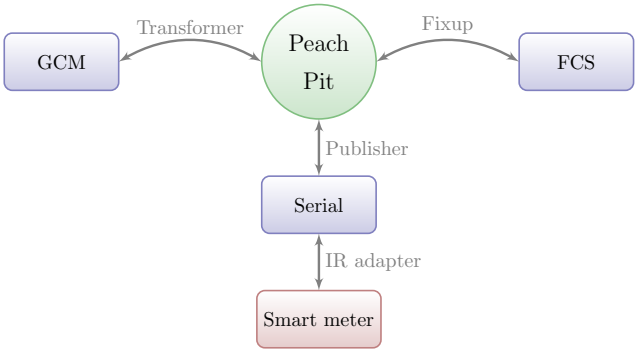


Figure 3: Different components used in eFuzz/Peach and their relationship.

As explained, the transformer is used to encrypt the mutated plaintext to the final form. In the current context, all necessary keys and passwords to encrypt and decrypt the text were available. Presuming access to this data can be acceptable or not depending on the context. For an "ordinary" attacker, it is not a realistic premise. Moreover, since every meter should be deployed with its own keys, in case of key disclosure only one device would be compromised. However, in the context of a security evaluation this is an acceptable assumption. For example, a company hired to assess the security of the device would be given access to the keys and perhaps the firmware source code. Then, they would use eFuzz in an initial approach to autonomously detect possible points of interest. The respective report would serve as a guide to security experts and complement a full featured source code review. Moreover smart meter manufactures can also greatly benefit from eFuzz by integrating it in their test suites, thus detecting flaws early in the product development process.

Generating packets

eFuzz uses a modular approach to generate the fuzzed requests. The most abstract data model is a common wrapper. The block incorporates the HLDC and the LLC layers. Some of the fields are randomly chosen while others, such as size and checksums, are computed deterministically. The checksum, also known as frame check sequence, is handled by a custom fixup instance. The DLMS payload is declared as well, but the actual contents are not specified.

There are two distinct models describing the APDUs sent from the PC to the meter.

1. The first APDU inherits the common wrapper and only replaces the necessary fields. The request is fuzzed at the plaintext level and consequently encrypted, through a user-defined transformer. This APDU is modeled after a request that queries the meter for the local time and date.

2. The second APDU defined by eFuzz is only used as a reference. It is similar to the previous one but none of the fields are modified.

The two APDUs that correspond to the answers from the meter serve different purposes.

1. The response to the fuzzed request is read but disregarded. The sole purpose of the first APDU is to provoke faults in the target therefore the response has little relevance.

2. The following response is used to assess the health of the meter. Since it corresponds to a reference request can be compared with a reference response. If the received and expected data do not match, eFuzz logs the event.

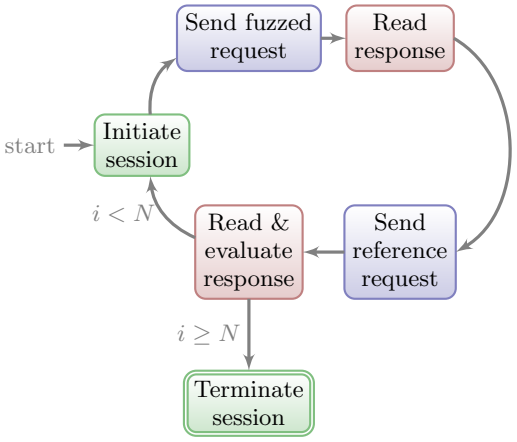
There is a myriad of strategies built-in Peach that define how the model is mutated. The existing mutation strategies are divided in two main categories: sequential and random.

As indicated by the name, sequential mutations are deterministic and progress from the first to the last fuzzable field. Each field is fuzzed only once. Within this category there is a distinction between linear (`SequentialMutationStrategy`), and random progress (`RandomDeterministicMutationStrategy`). These approaches are similar with only the order changing, which may allow the fuzzer to detect flaws earlier on the test suite.

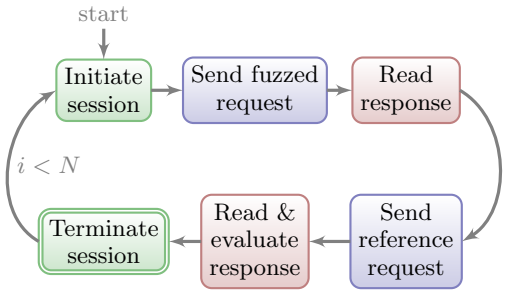
On the other hand, random strategies are not finite and fuzz up to N elements per iteration which allows for more flexibility and bigger coverage since combinations of fields are also fuzzed. The downside is the running time. Therefore, a typical test starts with sequential strategies and then proceeds to random ones. For reproducibility purposes, it is also possible to specify a seed for the internal random number generator.

In this category there are three mutators: `RandomMutationStrategy`, `SingleRandomMutationStrategy` and `DoubleRandomMutationStrategy`. The only distinction between the first and the others is the maximum number of elements that are fuzzed per iteration, which is 7 instead of 1 or 2. Nonetheless for `RandomMutationStrategy` the maximum value can be arbitrarily chosen. Furthermore, it is possible for developers to program their own strategies.

The response of the meter to the invalid queries is not necessarily relevant as there is not correct or incorrect response. More interesting is seeing how they might affect the meter behavior thereafter. Given the specificities of DLMS/COSEM two main ways were chosen to do so (see Figure 4).



(a) Fuzzed and reference packets are all sent within the same session.



(b) The session is restarted after each fuzzing attempt.

Figure 4: State machines describing the two approaches used to fuzz the target device. On these diagrams i indicates the current fuzzing iteration while N is the total number of tests available. In both situations the first iteration uses a reference request to ensure the target is in a known state.

The first approach is to send a reference request after each fuzzed packet. If the response matches what is expected, the meter is functioning properly. Otherwise, this behavior should be signalled and stored for human inspection. Thereafter the process is repeated. An exception to this cycle occurs when the target does not respond as expected. In this situation the session is restarted. Otherwise, the subsequent tests would not be evaluated by the meter as it was already in a faulty situation. As an example, assume request i causes the meter to reach an unknown state that rejects all further communication in the session. Then even if request $i + 1$ would normally trigger a new problem it would go unnoticed. The state machine describing this behavior is show in Figure 4a.

The second approach is similar but the session is restarted every time a fuzzing attempt terminates, as portrayed in Figure 4b.

These contrasting approaches may yield different results. The former is expected to detect more memory leaks or other memory related problems, for example, a memory leak triggered by an attack may not be mediately noticeable and only have consequences later on. If the session was terminated in between, there would be a chance for the meter to clear its state and recover from such a problem.

5. RESULTS

5.1 Detection Performance of eFuzz

To assess the performance of eFuzz, this section presents the amount of unexpected behaviors provoked in the target. As explained before, after sending each fuzzed request, eFuzz sends a reference query and compares the result with the correct response. If these values do not match, the dialog is stored as an unexpected behavior and the session is restarted before continuing.

Due to the detail used to model the protocol, the complete test suite of eFuzz when using the sequential strategies includes 54727 distinct cases. Running all tests requires around 7 days. However to be able to obtain data for different configurations, within a reasonable time frame, a subset of 500 tests was used. The subset completes within approximately 3 hours.

Table 1 shows the number of unexpected behaviors induced with three distinct strategies. Sequential Random refers to the aforementioned `RandomDeterministicMutationStrategy` class. The other two use random strategies where n specifies the maximum number of fields that are fuzzed per iteration.

Mutation strategy	**Number of unexpected responses**
Sequential Random	40
Random $n = 2$	21
Random $n = 7$	10

Table 1: Overview of the number of unexpected responses caused by eFuzz discriminated by strategy. Variable n indicates the maximum number of fields that are mutated in each iteration. In the sequential strategy this value is by definition one.

The amount of invalid responses is significant, in particular considering the short amount of time it required. The sequential strategy was the most proliferous, reinforcing the point made above that this strategy should be used first.

With respect to the random strategies there are several possible explanations for the lower number of unexpected responses. First, in these runs Peach includes iterations where it does not mutate any of the fields. Therefore it sends two reference requests consecutively, which results in a "wasted" iteration. A second reason, is that by increasing the number of invalid APDU fields per request, it is more likely that at least one of the boundary checks included in the code is triggered, causing the meter to ignore the request. This also helps explain why the increase in n results in less invalid responses.

In addition to the quantity, it is also insightful to understand what the meter response actually was. This information is condensed in Table 2. The responses are separated by exception-response, frame reject (specific DLMS/COSEM messages) or other.

The Exception-Response APDU is an optional APDU sent by the COSEM server application layer that indicates to the client application layer the service requested could not be processed. Particularly for the case shown, the request was not processed as access to the service was not allowed. Alter-

Mutation Strategy	Exception response	Frame Reject	Other
Sequential Random	50%	50%	0%
Random $n = 2$	57%	33%	10%
Random $n = 7$	50%	40%	10%

Table 2: Distribution of the unexpected meter responses by cause.

natively the server can simply discard the message without sending this APDU.

The Frame Reject (FRMR) is used to report one of various conditions occurred and retransmission of the identical frame cannot correct it. The situations that may trigger an FRMR include the receipt of an undefined or not implemented command or response, receipt of a frame that is too long, receipt of an invalid frame, or receipt of a frame containing a field that is not allowed.

Categorizing the meter responses shows a clear majority of `exception-response` and `frame-reject` APDUs. These messages are used to signal errors and should not occur in response to valid requests, which suggests one of the previous messages perturbed the internal state of the server and has caused undesirable side effects.

5.2 Discussion

The results presented in Section 5.1 indicate this preliminary work exposes bugs that would probably go unnoticed otherwise. Just like in traditional software application, fuzzing is a good technique to test proprietary code. The approach used can generate combinations of inputs that would not be possible by a human tester as such person would be inclined to follow the same assumptions as the original developers.

A series of experiments performed on the target were able to provoke between 10 and 40 incorrect responses, depending on the mutation strategy selected, in a short period of time. Although with this approach it is not possible to automatically determine if these bugs have security consequences, it pinpoints areas of interest that a human expert can focus on.

As stated in Table 2 the responses obtained from the target after fuzzing almost exclusively correspond to Frame Reject and Exception-Response APDUs. Although these responses were incorrect given the reference request they are valid protocol messages, and do not necessarily indicate the existence of security flaws. To further investigate the impact of fuzzing the target, lower level access to the hardware, or access to the source code, would be required. Given these conditions it would be possible to have a better understanding of the state of the software. Performing this type of analysis is suggested as future work.

Nevertheless, this method is not meant to replace professional human auditors but instead complement them. The added value is accomplished by means of an automated tool that exhaustively finds all possible bugs with respect to diverse inputs. Note that manual inspection by a human expert cannot achieve the same level of testing within an acceptable time frame.

To better understand the complementarity of both approaches, the results of eFuzz and a security evaluation made manually are compared. Table 3 compares the two with respect to four different parameters.

Parameter	eFuzz	Manual
Time	Hours/few days	Weeks
Coverage	DLMS/COSEM logical attacks	Physical attacks. Weaknesses in configurations, key storage, retrieval.
Quantity findings	High	Low
Quality findings	Low	High

Table 3: Comparison of automated and non-automated approaches to detecting smart meter vulnerabilities.

One of the main advantages of fuzzing is its automated nature. Automation makes it faster, hours or days instead of weeks, than manual evaluations and requires little human supervision. On the other hand eFuzz has a very specific coverage while a manual approach can be much broader and detect classes of vulnerabilities that a fuzzer cannot, for example if keys are stored securely or if permissions are properly enforced. As previously shown eFuzz is able to detect a large quantity of findings while manual inspections are typically more selective. This disparity is reflected in the quality of each discovery. Issues uncovered by fuzzing need to be inspected by an expert to determine if they constitute a real security threat.

6. CONCLUSION

This paper describes eFuzz, a DLMS/COSEM fuzzer built on top of Peach to complement security evaluations of smart meters. It works autonomously by generating invalid or malformed protocol packets derived from a model of the standard. By constantly sending these requests to the meter followed by a reference query, one can assess if the fuzzed data was able to interfere with the normal behavior of the target.

In the preliminary tests with a electricity meter currently in the market using a direct optical interface, eFuzz was able to perturb the correct functioning of the target. In less than 3 hours, the tests revealed between 10 and 40 issues depending on the mutation strategy. The majority of the invalid responses indicated that the client was trying to access a service that was not allowed or that the request was malformed, which suggests previous messages caused the meter to reach an invalid state.

The obtained results show that fuzzing approach is an efficient way to detect flaws in the protocol implementation given the limited human effort it requires. Additionally, our framework for fuzzing embedded systems, eFuzz, will provide researchers the capability to apply fuzzing techniques as a vital step for the evaluation of smart meters and other networked embedded systems. Ultimately, we hope that eFuzz will make smart meter communications harder to penetrate, and consequently elevate the bar for the security of critical infrastructures.

As mentioned previously, there is ample room for improvements in eFuzz. For example, in its current version only a small part of the DLMS/COSEM is modelled. To achieve better results, it would be beneficial to increase the coverage of the standard and consequently of the firmware code. Furthermore considering more parameters, such as the latency of the responses, when assessing the health of the meter is also a possible improvement. In addition, studying the impact of fuzzing by obtaining the source code running on the target or by inspecting the operations performed by the hardware components would provide greater understanding of the efficacy of eFuzz. Such insights would constitute a feedback loop to refine and improve the tool. Another advancement that is currently being worked on is exploring other communication interfaces, notably GSM, since not all meters are equipped with an optical port. Finally, at this time, the main fuzzers readily available are not optimized for embedded systems. As networking capabilities are introduced to a growing number of micro-controller based devices, it is fundamental to ensure security is preserved. Fuzzing can play a crucial role in this task, but first it needs to expand its capabilities to these new classes of targets.

7. REFERENCES

[1] D. Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 2002.

[2] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *Information Security*, pages 343–358. Springer, 2006.

[3] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding Software Vulnerabilities by Smart Fuzzing. *The Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 427–430, Mar. 2011.

[4] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A taint based approach for smart fuzzing. *The Fifth IEEE International Conference on Software Testing, Verification and Validation*, pages 818–825, Apr 2012.

[5] Deja vu Security. Fuzzing Embedded Devices with Peach Fuzzer. `https://www.youtube.com/watch?v=yevXIDaI_SA`. Accessed: 2014-06.

[6] Deja vu Security. Peach fuzzing platform. `http://old.peachfuzzer.com/v2/peach23.html`. Accessed: 2014-02.

[7] Deja vu Security. What is Peach? `http://old.peachfuzzer.com/WhatIsPeach.html`. Accessed: 2014-03.

[8] Z. Fan, P. Kulkarni, S. Gormus, C. Efthymiou, G. Kalogridis, M. Sooriyabandara, Z. Zhu, S. Lambotharan, and W. H. Chin. Smart grid communications: Overview of research challenges, solutions, and standardization activities. *Communications Surveys & Tutorials, IEEE*, 15(1):21–38, 2013.

[9] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke. A survey on smart grid potential applications and communication requirements. *Industrial Informatics, IEEE Transactions on*, 9(1):28–42, 2013.

[10] C. Liechti. pySerial's documentation. `http://pyserial.sourceforge.net`. Accessed: 2014-06.

[11] P. McDaniel and S. McLaughlin. Security and privacy challenges in the smart grid. *IEEE Security and Privacy*, 7(3):75–77, May 2009.

[12] S. J. McIntyre. Termineter. `https://github.com/securestate/termineter`. Accessed: 2014-06.

[13] A. R. Metke and R. L. Ekl. Security technology for smart grid networks. *Smart Grid, IEEE Transactions on*, 1(1):99–107, 2010.

[14] MicroSolved Inc. Protopredator. `http://microsolved.com/protoPredator.html`. Accessed: 2014-06.

[15] B. Miller. CS 736 Fall 1988 Project List, 1988.

[16] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[17] MWR InfoSecurity. Usb fuzzing for the masses. `https://labs.mwrinfosecurity.com/blog/2011/07/14/usb-fuzzing-for-the-masses/`. Accessed: 2014-02.

[18] A. J. Paverd and A. P. Martin. Hardware security for device authentication in the smart grid. In *Smart Grid Security*, pages 72–84. Springer, 2013.

[19] W. Simpson. PPP in HDLC-like framing. July 1994. RFC 1662.

[20] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. *IEEE Symposium on Security and Privacy*, pages 497–512, 2010.

[21] W. Wang and Z. Lu. Cyber security in the smart grid: Survey and challenges. *Computer Networks*, 57(5):1344–1371, 2013.