Security Test Approach for Automated Detection of Vulnerabilities of SIP-based VoIP Softphones

Christian Schanes, Stefan Taber, Karin Popp, Florian Fankhauser, Thomas Grechenig
Vienna University of Technology
Industrial Software (INSO)
1040 Vienna, Austria
E-mail: christian.schanes,stefan.taber,karin.popp,
florian.fankhauser,thomas.grechenig@inso.tuwien.ac.at

***Abstract*—Voice over Internet Protocol based systems replace phone lines in many scenarios and are in wide use today. Automated security tests of such systems are required to detect implementation and configuration mistakes early and in an efficient way. In this paper we present a plugin for our fuzzer framework fuzzolution to automatically detect security vulnerabilities in Session Initiation Protocol based Voice over Internet Protocol softphones, which are examples for endpoints in such telephone systems. The presented approach automates the interaction with the Graphical User Interface of the softphones during test execution and also observes the behavior of the softphones using multiple metrics. Results of testing two open source softphones by using our fuzzer showed that various unknown vulnerabilities could be identified with the implemented plugin for our fuzzing framework.**

***Keywords*-Software testing; Computer network security; Graphical user interfaces; Internet telephony; Fuzzing.**

I. Introduction

Voice over IP (VoIP) is in wide use in homes, educational institutions and businesses, extending or replacing Public Switched Telephone Network (PSTN) based phone lines. VoIP provides a way of sending phone calls over Internet Protocol (IP) based networks. This allows the use of one type of wiring for both computers and phones in office buildings, making management simpler and changes in the setup faster. However, moving from separate phone lines to commonly used IP based networks increases the attack surface and by this the risk for attacks. Therefore, a secure VoIP infrastructure is required where all components in the infrastructure are robust against attacks. This also includes the VoIP clients as shown in our previous work in [1].

Today, Session Initiation Protocol (SIP) is a widely used protocol to control communication between two VoIP components like initiation and termination of calls. It was introduced in 1999 by the Internet Engineering Task Force (IETF) in RFC 3261 [2]. It is a stateful text based signaling protocol, which defines two main components for communication: the User Agent (UA) and the Server. A UA can be a soft- or hardphone and initiates or terminates sessions. A Server offers services to UAs, e.g., to register and to relay calls. Phones are reachable by other phones to allow communication and, therefore, a vulnerability within VoIP phones can be remotely exploited by attackers. Our approach shows the possibility to test VoIP phones by using simulated attacks to detect vulnerabilities to make the software more robust.

Fuzzing as a dynamic method to detect vulnerabilities by fault injection was used early by Miller et al. [3], [4] to detect security vulnerabilities in different applications. Since then, fuzzing has become a widely used method to test software robustness and security of different applications [5], [6]. In this paper we focus on testing SIP interfaces of Graphical User Interface (GUI)-based UAs using fuzzing techniques to automatically detect security vulnerabilities by monitoring multiple interfaces of the UAs. For this we present an extension for the fuzzer framework fuzzolution [7].

Thompson [8] defines security failures as side effects of the software which are not specified and make security testing hard. Fuzzing provides a solution for detecting side effects by automatically executing test cases with many data variations based on critical well known attacks and randomly generated values. The introduced fuzzer framework supports the rules presented by Chen and Itho [9] to generate SIP test data. Therefore, an intelligent template based approach [10] with random attack data generation and predefined well known attack values is used. Additionally, with the used state machine it is possible to test different valid and invalid SIP states of softphones.

Automatically triggering GUI events, e.g., accepting or rejecting VoIP calls, is required to test SIP-based softphones because automatic GUI interaction makes it possible to explore a significant portion of the state space. The framework supports all kinds of mouse and key events to interact with the GUI. To improve accuracy of error detection in our approach we monitored multiple interfaces of the GUI-based softphone. One used metric was a GUI monitor on the client side, which allows for detection of error dialogs and changes in the application windows. Other used metrics are log files of softphones, text analysis of the content of open windows on the client side, analyzing the response, monitoring CPU and memory usage and monitoring the availability of the application by using the network ports of the softphones.

We present the results of our proof of concept approach applied to two open source softphone implementations. We found several previously unknown vulnerabilities. We also tested older releases and found known vulnerabilities, which have already been fixed in newer versions.

The remainder of this paper is structured as follows: Section II discusses related work. Section III introduces the architecture of the test environment and the used fuzzing framework. Section IV gives details about the implementation of test generation and the monitoring of the System Under Test (SUT) to automatically detect errors. The implemented fuzzing framework and detection techniques were evaluated by testing SIP implementations in a VoIP test environment, which are presented in Section V. The paper finishes with a discussion in Section VI and a conclusion and ideas for future work in Section VII.

II. Related Work

Various partly overlapping technologies have been introduced to cover different aspects of VoIP calls, e.g., signaling standards (that take care of the setup of a voice channel). Several signaling technologies are in use today: H.323, Media Gateway Control Protocol (MGCP), SIP as well as proprietary solutions, e.g., InterAsterisk eXchange (IAX) protocol. SIP is probably the most widely adapted protocol [11] and, therefore, in focus of our presented security test approach. As shown by different authors (e.g., [12]–[14]), many attacks against SIP implementations are readily available and conducted in deployed VoIP infrastructures.

Fuzzing is a test technique to find vulnerabilities in different applications [3]–[5] and, therefore, also in SIP-based VoIP applications [15], [16].

Several frameworks have been proposed to address specific aspects of SIP security, for example, the PROTOS SIP Test Suite [15], [17], [18], which has more than 4500 predefined malformed SIP-Invite messages to test the robustness of SIP implementations. However, it only supports stateless testing of SIP phones, which reduces the possibility to test further SIP states like Calling or Ringing. Additionally, PROTOS does not support client testing by triggering GUI actions. Another approach is the one followed by Aitel with SPIKE [19] which introduces the concept of block-based fuzzing. This is based on the fact that protocols are mostly composed of invariants, blocks and variants. SPIKE fills the blocks with fuzzed data and keeps intact the invariants. SPIKE is too low level, so it becomes highly effort-consuming when applied to complex protocols. Banks et al. [20] described SNOOZE, which is a generic fuzzer framework based on user defined scenarios and protocol specifications. Currently only a limited number of primitives to generate malformed data are implemented. However, the authors showed how to use SNOOZE to fuzz SIP implementations and find security vulnerabilities. Another stateful fuzzer is KiF, described by Abdelnur et al. [16], [21]. KiF uses Augmented Backus Naur Form (ABNF) grammar to describe the syntax of messages. KiF automatically generates new crafted messages by using rules defined by the grammar, information of the current protocol state and state tracking information. The authors showed how to test different states of SIP. A drawback is the analysis for detecting errors, which is only based on the responses of the SIP implementations. Nevertheless, with the described approach several vulnerabilities were found. This includes bad input handling and state based vulnerabilities, where the robustness of the implementation failed by using various valid/invalid state paths.

Alrahem et al. [22], [23] presented a fuzzer with the name INTERSTATE. The fuzzer provides the possibility to interact with the GUI of softphones and, therefore, allows, for example, to automatically accept calls. During test execution the fuzzer can send sequences of SIP messages and GUI events to the SUT, which provides the possibility to test a larger range of implemented states of softphones automatically. In contrast to the approach provided in this work, however, the analysis to detect errors is only based on the response and does not consider the behavior of the softphone GUI.

White-box fuzzing is another approach of using internal information of an implementation as, for example, presented by Ganesh et al. [24], by Godefroid et al. [5], [25] or by Neystadt et al. [26]. However, it is not always possible to get the internals of an implementation. For example, if hardphones are being tested, the software is often closed source and only black-box tests are possible. Therefore, we focused on implementing black-box tests for our approach. These can be used for testing a broader number of VoIP clients.

GUI automatization is already used for functional testing of applications as, for example, presented by Feng et al. [24] or by Xiaochun et al. [27]. Bo et al. [28] presented a black-box test approach for mobile phones using Optical Character Recognition (OCR) to analyze the GUI behavior. To improve automatization for testing SIP-based softphones, interaction with the GUI of phones is required as already shown by Alrahem et al. [22], [23]. In our implementation we integrated the automatic GUI interaction. Additionally, we present a basic approach for observing GUI behavior and using the gathered information to reduce the false-negative rate.

III. Architecture for Automated VoIP Softphone Testing

The implemented fuzzer is a generic framework to test and monitor different application interfaces similar to other fuzzers. For testing SIP-based softphones, a template based message generation method was chosen. The fuzzer includes

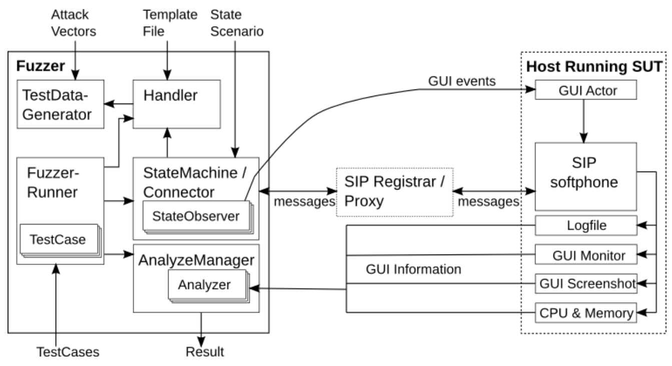

Figure 1. Test Environment of Fuzzer Framework for SIP Softphones

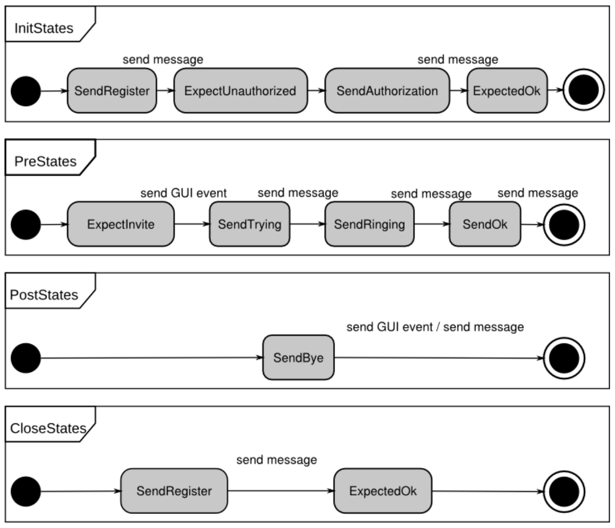

Figure 2. Example of a State Definition

a configurable SIP state machine which allows stateful testing of phones by interacting with the GUI.

Fig. 1 shows the test environment and the interaction between the fuzzer and the SUT. The fuzzer provides the possibility to automatically control and monitor the SUT by its GUI using a GUI Actor and a GUI Monitor. The state machine can trigger GUI events in the SUT using the GUI Actor. Additionally, the GUI Monitor provides information about the GUI to the analyzer which integrates the information to determine pass/fail of tests.

The test environment at the host of the SUT consists of the components softphone (SUT), GUI Actor, the GUI Monitor, the log files produced by the SUT, CPU and memory monitoring and a screenshot component as can be seen in Fig. 1.

A. *Fuzzer Framework*

As framework for executing SIP softphone tests the fuzzer framework fuzzolution [7] was used. This framework provides the possibility to add plugins for specific protocols. We implemented a state based SIP plugin and extended the framework with various analyzers to monitor the behavior of the SUT.

The main parts of the fuzzer framework, as can be seen in Fig. 1, are the `FuzzerRunner`, which controls test execution, a `Handler` to generate the messages to send, the `Connector`, which includes the protocol to communicate with the SUT, the `AnalyzeManager`, which builds the final test result based on all controlled `Analyzers`, and a set of `TestDataGenerators` which provide the used test values.

The implemented `StateMachine` for SIP contains the SIP states and the interaction with the GUI and the SIP interface of the SUT. The framework is independent about stateful or stateless connector implementations. The used `StateMachine` for SIP inherits from the `Connector` type which allows transparent integration into the framework. Based on a configuration file it is possible to configure the various SIP state transitions. Additionally, it supports configuration of invalid state transitions to test the behavior of the SUT by using unspecified transitions. The state machine has a sequence of states to execute during initialization, before sending the test data, and after sending the test data.

The `StateMachine` establishes a connection either directly with the SUT or via a proxy, sends generated SIP messages and receives responses from the proxy or the SUT. Additionally, `StateObserver` can be registered to the `StateMachine`. The `StateMachine` notifies all registered `StateObservers` before and after state transitions. By the usage of the `StateObserver` additional functionality can be executed, e.g., to interact with the GUI of the softphone.

The `Handler` is controlled by the `StateMachine` and is responsible for the generation of valid as well as malformed messages. Additionally, information from the state machine can be used to generate different parts of the message, e.g., the ID. Multiple handler implementations are used for test execution depending on the tested parts of the SUT.

For generating test data various `TestDataGenerators` will be used. It is possible to extend the framework by implementing customized generators. Different implementations are used for generating the test values. They are based on well known attack vectors, random generation and intelligent generation. For more details about generation of test values see Section IV.

For a test case the configuration of state transitions and GUI interactions is called a state scenario. State scenarios are integrated in the fuzzer framework at different execution points: before and after the test run and before and after a single test case. Based on the managed state information, the

`StateMachine` prepares the SIP softphone, i.e., it brings the softphone into the required SIP state for a test case. For example, a valid registration state scenario can be defined before the test run to register the fuzzer to a SIP registrar and, after the test run, cleanup by de-registering the fuzzer.

Fig. 2 shows an example of a state configuration where as init states the fuzzer registers to a SIP registrar (in the test environment to the proxy). Therefore, the `StateMachine` sends a Register message and expects receiving an Unauthorized message. The `StateMachine` resends the message together with a valid authentication header and finally expects an Ok message. After this initial registration for each test case the fuzzer will execute the `PreStates` before sending the test data and the `PostStates` after sending it.

A state scenario before a single test case can be used, e.g., to initiate a call from the SUT to the fuzzer by sending a GUI profile to the GUI Actor which initiates the call by executing the GUI profile.

B. Simulating the SIP Registrar

SIP can establish a call directly between two UAs but many SIP phones are registered on a SIP registrar and send all responses to this registrar which acts as proxy and forwards the messages to the destination UA.

When using such a proxy, it is required that the proxy does not filter invalid SIP messages. Therefore, we implemented a specific SIP registrar and proxy as part of the framework which is robust and forwards the manipulated test messages to the SUT without modifications.

The used version of the implemented SIP registrar supports the used SIP scenarios. It accepts all registration requests of any UA and internally stores the registration information, e.g., the username and the address of the UA for further communication.

The SIP registrar supports several parallel UAs. It allows forwarding SIP messages between every registered UAs without further checks. The proxy determines the receiver of the message by the `callid` or by the username in the `To` header field or `Request` header of the incoming message automatically. Additionally, it is possible to manually configure the host and port of the two communicating UAs – in the test setup the fuzzer and the SUT. This also allows to test the `To` and `Request` header.

Requests to UAs which are not registered will be handled by sending a `Not Found` message to the requesting UA. The proxy will respond to every request addressing the proxy with a valid Ok message which occurs often for `Option` messages.

C. GUI Interaction with the Softphone

The utilization of GUI events is essential to automatically achieve all possible states of the SIP softphone. The provided framework supports the definition of input sequences with SIP messages and GUI events, which are remotely applied to the SUT.

On the side of the SUT an additional service called GUI Actor is running. The GUI Actor is a Java tool, which uses the `java.awt.Robot` implementation to interact with GUI elements by executing GUI events in the SUT. Using the GUI Actor, automated test execution for GUI-based SUTs is possible.

The fuzzer communicates via network with the GUI Actor using GUI event profiles which are part of state scenarios. A GUI event profile is a sequence of GUI events and an unique identifier to identify the profile. In the current version, mouse and keyboard events are supported. Delays in the sequence can also be configured. This is, e.g., required to wait for opening dialogs in the application before interacting with the GUI.

During initialization the fuzzer registers all required GUI event profiles with the unique identifier at the GUI Actor running in the SUT. For the profiles simple text files with key/value pairs are used. The key is the action, e.g., left mouse click or key press, and as value specific parameters for the action are used, e.g., coordinates for clicking or specific keys to press.

The GUI Actor can handle several profiles simultaneously. By using the profile identifier the fuzzer can execute every registered profile by sending the execute command and the profile identifier to the GUI Actor.

D. Observing System Under Test Behavior

To reduce the false-negative rate, the introduced framework uses multiple metrics of the SUT to build the final test result. For this the fuzzer framework supports the combination of various `Analyzers`.

One analyzer was used to verify if the arrived response from the SUT is valid (e.g., the expected message for this state transition) and if it contains specific keywords indicating a failure, e.g., `Exception` or `Error`. A second analyzer monitors the SUT by verifying if the port of the application is available after executing a test. If the port is not available it indicates a possible crash of the application. Both analyzers can be used for black-box tests without access to the host where the SUT is running.

An analyzer was used to analyze the log files of the SUT and determine important log entries using keywords. For this the log files will be transferred using a Secure SHell (SSH) connection. The analyzer uses only the entries produced during executing the test case by building the delta of the log file before and after the test execution. The analyzer uses the detected log entries and based on a predefined list of weighted critical keywords returns the build probability combining the number of keywords and the associated weight.

Another observed behavior of the SIP softphones is the GUI. Applications visualize error information in different elements of the GUI, e.g., dialogs. The analyzer monitors the GUI of the SUT and identifies changes of open windows in the host after the test case in comparison to the windows before executing the test case. For all newly opened and closed windows, the analyzer estimates the relevance by using the name of the title and relation of the window to the SUT. The relevance of a window in the SIP softphone is higher than the relevance of other applications in the host. Examples of changes are new open error (or similar) dialogs or disappeared windows, e.g., after a crash of the SUT. Both analyzers require access to the host running the SUT.

In addition to the window structure the framework uses the visualized text information for error detection. For this OCR is used for a screenshot of the SUT to extract the text information displayed on the screen. The screenshot will be taken before and after executing the test case. With OCR the text will be extracted and analyzed using a list of critical keywords. The screenshot analyzer can, therefore, also detect errors displayed directly in the main window without opening a new sub window.

The framework also analyzes CPU and memory usage of the softphone process in the SUT to detect heavy load during/after executing a test which could indicate a Denial of Service (DoS) attack.

The GUI Monitor, the screenshot analyzer, softphone log file analyzer and CPU/memory monitor require access to the SUT to start a service to transfer the information to the host running the fuzzer. Both services wait for a connection from the fuzzer and the fuzzer has to trigger the services to get the information.

IV. Framework for Implementing Security Test Cases for SIP-based VoIP Softphones

The fuzzer framework with the SIP plugin provides various possibilities to configure tests for SIP implementations and especially for GUI-based softphones. Within the fuzzer framework the following notation is used for the test configuration which also builds the further structure within this section:

Test Scenario: the scenario is a specific sequence of state transitions, e.g., send an Invite message to the SUT. We executed the fuzzer for each scenario.

Test Case: we define the different fields within a single message as test case. These fields will be replaced by generated values, e.g., an Invite message has several fields and each field indicates one test case.

Data Variation: data variations are the different values used for filling the fields in test cases. We use many data variations for each test case.

Chen and Itho [9] describe five rules that can break robustness of SIP-based VoIP systems by manipulated messages:

- Incorrect Grammar
- Oversized Field Values

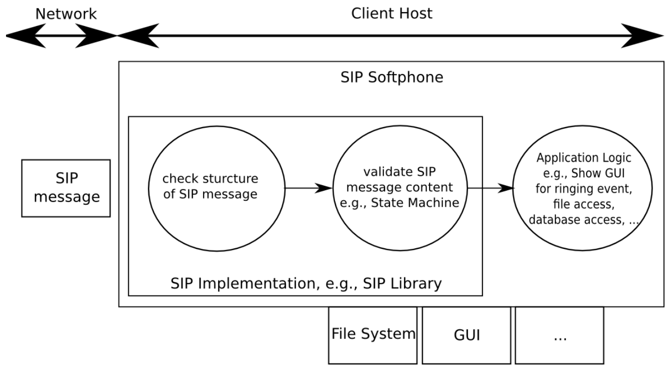
Network
Client Host
SIP Softphone
SIP message
check sturcture of SIP message
validate SIP message content e.g., State Machine
Application Logic e.g., Show GUI for ringing event, file access, database access, ...
SIP Implementation, e.g., SIP Library
File System
GUI
...

Figure 3. Generic SIP Message Parsing and Processing by SIP Implementations

- Invalid Message or Field Name
- Redundant or Repetitive Header Field
- Invalid Semantic

All of these five rules are considered for testing the SIP softphones with the defined test scenarios, test cases and data variations in our test setup.

Analyzing the attack surface of an application is important for conducting security tests. The attack surface shows possible interfaces with associated risks for exploiting vulnerabilities in the interfaces. Fig. 3 shows a general attack surface for SIP-based softphones. For the current work we considered remote attacks using the network interface of the application. The figure also shows relevant modules within the application which are involved in processing SIP messages.

A. *Test Scenarios for SIP*

The fuzzer provides the possibility to configure different test scenarios for testing SIP-based VoIP softphones. Each scenario is a single test run of the fuzzer tool. In the current version of the implementation we defined different scenarios to show that the proof of concept of GUI interaction and monitoring of the SUT was working. The analysis is based on various defined SIP scenarios as presented by Johnston et al. in RFC 3665 [29].

Fig. 4 presents the main test scenarios used for the test execution. The first scenario presented is stateless and has already been tested repeatedly using different fuzzers as, for example, PROTOS. The fuzzer sends malformed Invite and malformed Cancel messages to the SUT. Many DoS attacks are based on such a scenario, because no additional conditions are required, e.g., a valid SIP account, to send a message to the SUT. The second scenario represents a typical SIP call flow where the fuzzer calls the UA (SUT). The call flow is a RFC conform state transition. The fuzzer uses the final ACK message to construct test messages. In comparison to the second scenario, in the third scenario the SUT initiates the call. The fuzzer triggers this by the GUI Actor and the SUT calls the fuzzer, which replies with a Trying, Ringing and Ok. The fuzzer terminates the call without waiting for the required ACK message from the SUT by sending a Bye message. Each message sent by the fuzzer in this scenario is used for the test execution to include the test data. This scenario tests the robustness during the call initiation phase of the UA. The fuzzer initiates a call in the fourth scenario. After receiving the Ok message from the SUT the fuzzer sends a manipulated Cancel message. Similar to the third scenario the SUT calls the fuzzer, which responds with a valid Trying and Ringing. After that the fuzzer sends a malformed Unauthorized message. This Unauthorized message is not expected by the SUT. Therefore, we test how the softphone deals with unexpected and manipulated messages.

For all scenarios except the first one, access to the GUI of the SUT is required for automatic testing in order to initiate, accept and close calls.

The fuzzer supports valid scenarios with RFC conform state transitions and invalid scenarios. Valid scenarios are required to bypass message validation functionality to inject the test values in the application logic as can be seen in Fig. 3. If the validation is not successful the message will be rejected and testing of the application logic is not possible. With invalid scenarios the validation logic within the SIP softphone can be tested.

B. Test Case Design for SIP

Test cases in the fuzzer framework are messages which are part of a test scenario. For the tests a template based approach was used. The template defines the message structure and defines placeholders which indicate the single test cases within a test scenario. The `Handler` within the fuzzer framework is responsible to prepare the final message by replacing the placeholder of the current test case with the attack value of the data variation. All other placeholders are filled by using valid default values which can be predefined static values or generated values, e.g., unique identifiers or timestamps. The `Handler` must also support handling specific values, e.g., the `Content-Length` header. Otherwise the UA will eventually drop parts of messages.

The messages are constructed according to the SIP RFC to provide a valid structure and only use test data within the fields to test application logic as can be seen in Fig. 3. Moreover, tests with invalid structures are used, e.g., randomly modified orders of SIP headers, duplicate, randomly injected characters or invalid header fields. These tests are required to test robustness issues of the SIP message validation functionality.

The fuzzer framework supports cascaded `Handler` configurations, e.g., use the template handler to prepare the message and afterwards use a handler to manipulate message structure.

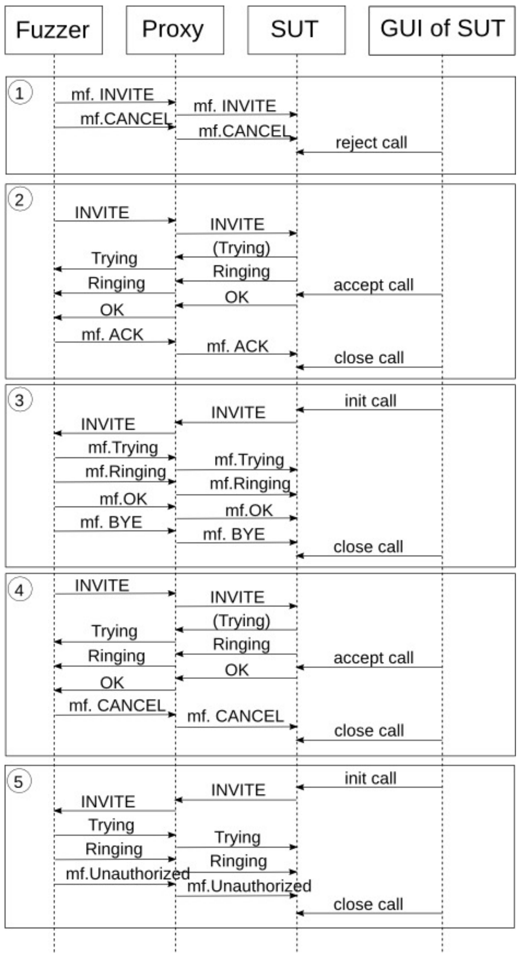

Figure 4. Definition of Used Test Scenarios for SIP Softphone

C. Automated Test Data Generation

Several test cases with several thousand data variations are defined based on the rules mentioned by Chen and Itho [9]. For the applied fault injection approach well known attack vectors and randomly generated data can be used. For the tests an optimized list with about 1200 known critical values as attack vectors are used. This includes various security attacks, e.g., buffer overflows, Structured Query Language (SQL) injections and path traversal attacks.

Additionally, generators are used to generate test data. The fuzzer framework allows integration of additional generator implementations to test specific values. For the test execution randomly generated bytes with random length are used.

D. Automated Error Detection

The fuzzer framework uses multiple analyzers to determine pass/fail of tests. Different analyzers are used for testing the softphones and with each analyzer it is possible to detect different vulnerabilities. For coordinating the results of these analyzers an `AnalyzeManager` is used which collects the results of all analyzers to calculate the final result for one data variation by summing up the results. In this step it is also possible to change the weight of the results based on predefined rules.

Before starting with the tests the fuzzer initiates a learning phase. In this phase valid data variations are used to determine how the application behaves with valid requests so that differences can be detected when using attacks. During the learning phase, analyzers store information about the found error indicators, so that they can be seen as normal behavior of the SUT and not used for further error detection. For example, the log file analyzer stores the information of how many times each keyword can be found for a valid request in the learning phase. The analyzer only uses keywords for the test execution which occur more often than the initially learned number of found keywords in the log file. This is required because depending on the error handling keywords like ”Exception” can also be included in the log file for valid requests.

Several analyzers are used to monitor the SUT as described in Section III-D. The port analyzer checks if the port, where the SIP softphone is listening, is still available by connecting to the port. The weight of this analyzer is very high and indicates that an error was detected certainly when the application cannot be reached any more.

The GUI analyzer uses the list of open windows in the SUT. We implemented the GUI Monitor prototype running on the SUT. The GUI analyzer retrieves the required information by a network connection from the GUI Monitor which uses the `xwininfo` utility for X to read window information. In an operating system a tree with one root window is available and every other window has exactly one parent window. Our GUI Monitor parses the output from `xwininfo` and stores the information in a tree structure. For each test case this is done before and after executing the test case.

In addition to the list of open windows we used an OCR analyzer which extracts the displayed text from a screenshot of the host desktop and uses a list of keywords for error detection. As OCR engine we used the `tesseract` command line tool. For taking the screenshot a service is running on the SUT. The analyzer requests the image using a network connection and extracts the text using OCR. Example keywords used for the test execution are: ”exception”, ”error”, ”wrong”, ”fault”, ”failure” or ”debug”.

With a response analyzer the fuzzer analyzes the response of the application and checks if certain keywords can be found. The keywords are stored in a configuration file as pairs of keyword and weight of the keyword. The analyzer checks for each keyword if it is included in the response and sums up all weights of the found keywords.

The log file analyzer is very similar to the response analyzer, but uses log files instead of responses. The analyzer determines the delta of a log file to use only log entries logged during execution of the current test case. Example keywords used for the test execution are: ”NullPointerException”, ”IOException”, ”Exception”, ”Bad”, ”Missing”, ”error”, ”fatal” or ”segmentation fault”.

The CPU and memory analyzer monitors the softphone process in the SUT. During the learning phase the analyzer determines CPU and memory usage for valid requests. For each test the analyzer determines the variance to the learned behavior. Depending on the implementation increased CPU or memory usage is possible due to internal reorganization activities, e.g., reorganize or cleanup a cache.

The `AnalyzeManager` gets the probability of a detected error from each analyzer. Each analyzer has a configured weight and based on defined rules the analyzer manager combines the final result using the computed probability of the analyzers and the weights of the analyzers for the final result.

Additionally, the analyze manager performs a final grouping of the result which supports possible required manual checks and retests. The grouping is based on the assumption that test cases with the same probability value have also similar log file entries, similar response values, similar GUI behavior etc. The group contains the list of test case IDs and the detected problem, e.g., a specific exception in the log file. For manual verification of the result to determine if the reported error is a true error, often only one test case per group has to be retested. Furthermore, the higher the result value of the group the more likely it is, that the reported error is a true one. Currently the implementation determines pass/fail based on a configured threshold of the combined probability.

V. Results of the Automated Test Approach

The fuzzing framework fuzzolution was used to test two SIP-based VoIP softphones. This section describes the softphones (SUT), the used VoIP test setup and the detected vulnerabilities. A description of the evaluation of the approach is also included.

For the test execution the test scenarios defined in Section IV were used. With each scenario various unknown vulnerabilities could be identified. These include DoS, memory corruption, improper validation of array index, use of uninitialized variables and a kind of Cross Site Scripting (XSS) vulnerability in both tested softphones.

The test execution showed that every implemented analyzer has its strengths and weaknesses. The combination of the analyzers significantly improves the overall test results.

A. VoIP Test Setup

For the test execution the VoIP test environment as described in [30] has been used. The environment supports various requirements for executing security tests:

Flexibility: Different virtual images provide basic functions, e.g., a time server, a SIP registrar, Domain Name System (DNS) server etc. which can be used as a basis for the test setup. This way we could concentrate on configuring the relevant aspects for security testing the VoIP softphones with the fuzzing framework which was the objective of this work.

Scalability: The VoIP test environment is scalable. Fuzzing softphones needs a lot of resources to test multiple attack vectors. Especially for GUI-based applications execution time is increased because display of the GUI element and executing the action is asynchronous. To reduce execution time, multiple virtual client images were executed in parallel for the conducted tests. This way the execution time for multiple test scenarios could be significantly reduced.

Easy Analysis: A version control system is implemented to provide an easy mechanism for managing, e.g., the tested software versions, used configuration files, log files or network dumps. This helps analyzing security failures found during the security test process and provides a mechanism for retesting test cases in case of found security vulnerabilities.

Automation: Many aspects of the test process can be automated. Therefore, the needed test setup for executing the security tests can be quickly achieved when doing different tests.

Moreover, we used the capability of the test environment of capturing the network traffic. This allowed detailed analysis of the transferred messages.

During test execution two instances of the two tested VoIP softphones (see Section V-B) were used. The whole process of starting logging and network dumps, starting the softphones, GUI Actor/Monitor, the fuzzer etc. was automated by using different scripts. After the test execution the relevant data (e.g., log files of the VoIP softphones, output of the fuzzer framework) were automatically collected and added to the version control system.

For correlating and analyzing multiple events and log files a common time source was needed. This was achieved by using the time server provided by the test environment. This way timestamped events in the test environment, even if produced in different parts of the test environment (e.g., fuzzer and SUT), could be combined easily and, e.g., fuzzed input strings could be correlated to malfunction of the softphones.

The management interface of the test environment was accessible remotely. This enabled starting and stopping test cases efficiently.

B. Tested SIP Softphones

Two different softphones were tested. The first softphone is QuteCom [31]. It is an open source SIP implementation written in Python, C and C++. We mainly tested the version 2.2 revg-20100116203101 with our fuzzing framework presented in [1]. With the extended framework we tested version QuteCom 2.2 revg-20101103220243. The second softphone is SIP Communicator [32]. It is an open source Java VoIP and instant messaging client. For the work presented in [1] we tested the version 1.0-alpha3-nightly.build.2351. For the extended framework presented in this paper we tested SIP Communicator 1.0-alpha6-nightly.build.3189.

Both applications are available for Windows and Unix platforms. Linux Ubuntu was used as the host system running the SUT for the test execution. Tests during development of the fuzzer framework also showed the possibility to identify vulnerabilities for a previous version of QuteCom on Windows (without using the GUI monitoring) where QuteCom failed to check the content-length SIP header. QuteCom crashed every time when the content-length was less than the real content-length. When the current version of QuteCom was tested this vulnerability had already been fixed. In a future work the GUI interaction and monitoring will also be implemented for Windows operating systems.

C. Detailed Test Results

By implementing the fuzzer framework it was possible to detect vulnerabilities for both tested softphones. The vulnerabilities were reported to the developers. A DoS vulnerability in SIP Communicator could be identified. SIP Communicator had implemented a fixed source port range between 5000 and 6000. It does not reuse port numbers and, therefore, for each connection a new port number will be used. After 1000 used ports no additional calls could be handled by the application. The error was detected by the GUI analyzer, because the application showed an error dialog but did not crash. Therefore, the process of the application is available and it could not be detected by monitoring the processes. Monitoring the availability of the port of the SUT will not register a fail either, because the problem was only for connections from the SUT to another host. Another solution to detect such an error is by using a valid use case which will fail if the application is not available anymore.

The problem was already fixed in the current version of SIP Communicator but the fuzzer identified a similar bug in the new SIP Communicator version. Again, only a range of ports is available and with many open ports the application logs an exception to the log file. The log file analyzer detected this problem. Moreover, it was also detected by the GUI analyzer because SIP Communicator opened additional dialogs. The vulnerabilities were detected, because due to the automation of the test process it was possible to send many requests, which causes these problem.

Another problem in SIP Communicator could be identified during the monitoring of the application log files. The application logs many `java.lang.NullPointerException` for different manipulated calls. This problem could not be identified using the GUI monitoring, because the GUI does not display the error information. We also detected several `java.lang.ArrayIndexOutOfBoundsException` errors. The log analyzer reported this as error because the keywords are contained in the log messages. Additionally, also the GUI monitoring detected the error because it was a problem with open dialogs which are not closed anymore.

Additional tests of SIP Communicator showed further crashes of the application. The GUI analyzer has identified the problem, because the process of the application terminated and, therefore, all windows of the application disappeared. This was a change of windows, which the GUI analyzer interpreted as an error in the application and, therefore, the test case failed. Such a termination of the process could also be identified by monitoring the process or by trying to connect to the port of the application. Retests of single test cases could not reproduce the problem. For future development the fuzzer should automatically retest different combinations of test cases of a test run to automatically identify such problems.

The fuzzer framework identified a type of XSS vulnerability because SIP Communicator uses `javax.swing.JOptionPane` for constructing error dialogs, which uses `javax.swing.JLabel` to render the text. By directly sending an Invite message to the SUT with a manipulated From SIP header it was possible to inject HTML code, which was displayed by the JLabel. This did not allow the injection of JavaScript code but it was possible to create a connection to a remote host and download an image by using an `<img>` element. In combination with a Cross Site Request Forgery attack this could compromise the internal network of the user. This error was detected by the GUI Monitor because by the injection of Code the displayed dialog has a different size and, therefore, was not closed by the used GUI Actor.

QuteCom is also affected by a DoS vulnerability, which crashed the application. The SUT calls the fuzzer, the fuzzer accepts the call and sends a Bye message to terminate the call. Additionally, the fuzzer uses the GUI Actor to click on the terminate button in the SUT, which causes the crash in the application. The problem was detected by the GUI analyzer and the port analyzer.

Several memory corruption vulnerabilities could be identified for QuteCom, which stop the application with a segmentation fault. One error was identified with multiple parallel calls in Ringing state. Closing the ringing dialogs crashed the application. Another error could be identified by opening several parallel calls. QuteCom shows the error message ”insert number for forwarding”. If the fuzzer triggers the logout button the application crashes. The crashes were detected by monitoring the port of the SUT and by the GUI Monitor because if the application crashed the list of open windows changed. This vulnerability showed that in future tests additional scenarios for testing parallel interaction should be defined. During development we also detected a DoS vulnerability if two separate accounts are configured. QuteCom regularly sends Option messages which the used SIP Registrar answers with an Ok message. QuteCom crashes directly with this behavior. For future work it will be required to integrate further scenarios including several registrations and parallel tests.

QuteCom crashed after sending a message with a manipulated Length field. The error was detected by using the port analyzer because the port is not available anymore after the crash of the application, GUI Monitor because the window of the softphone closed and CPU usage because the process terminates.

During testing SIP Communicator we also detected a problem in the GNOME implementation. The applet displaying an incoming call caused high CPU usage in the system. We detected the problem because the tests take quite longer than other tests. This problem showed that for future tests, in addition to monitoring the CPU and memory usage of the softphone process, also the CPU and memory of the system should be analyzed because a misbehavior of the softphone could also lead to an unstable system.

VI. Discussion

Evaluating the performance of a fuzzer is a difficult task, because only the detection of new vulnerabilities is measurable. Without a benchmark with known vulnerabilities, further analysis is not possible. The fuzzer detected vulnerabilities, which were analyzed manually to evaluate the result of the fuzzer.

The results showed that the implemented fuzzer framework could identify vulnerabilities in SIP-based softphones. The combination of multiple analyzers for a test execution is essential to automatically detect vulnerabilities and reduce the false-negative rate. The implemented combination of the single analyzer results with the initial learning phase and the weight for each analyzer improved error detection compared to the results presented in [1].

Multiple detected vulnerabilities showed that interaction with the SIP softphone via its GUI is essential to automatically execute security tests. Some problems could only be identified by interacting with the GUI.

The analyzers detected different vulnerabilities. The accuracies of analyzers are different depending on the measured property. The analyzer monitoring the port detects failures with high accuracy. Only network problems caused false-positives, e.g., if a firewall blocks required ports.

Analyzing the responses based on a keyword table, i.e., verifying if a response contains specific keywords indicating an error, has not produced reasonable results. The quality of using this information for automated testing strongly depends on the implemented error handling. In the tested applications no relevant information for the analysis was sent in the SIP response.

Analyzing GUI behavior showed that this can be an important information for detecting vulnerabilities. Some of the identified vulnerabilities were only detectable by such an approach. The used OCR analyzer did not detect vulnerabilities but the produced screenshots were helpful for verifying the produced test results. For future work the OCR analyzer should be improved to increase detection of the characters within the image.

In the work presented in [1] Asterisk was used as a proxy in the test environment. We encountered problems with some messages, where Asterisk filtered messages and, therefore, Asterisk was the test target and not the intended SUT anymore. For the tests presented in this work we implemented a special proxy, one which forwards fuzzed data without modification.

By analyzing the fuzzer output, we identified significant differences in the execution time of test cases within a test run. We further investigated this finding, but could not reproduce the results. By using timing as an aspect of an analyzer during automated tests, it is required to use special test environments which do not falsify timing properties. In the used environment this requirement was not fulfilled.

For future work, the performance of the fuzzer should be improved. Especially the GUI interaction should be enhanced, because currently it is required to define delays in order to allow the application to open windows, e.g., to start a call. Minimizing the delay could improve the runtime performance of test executions. For optimization we used a test setup where multiple instances of the softphones are tested parallel by the fuzzer. However, our focus for the current implementation was the automatization of vulnerability detection as opposed to minimizing execution time.

VII. Conclusion and Future Work

This paper presents an automated security test approach, which increases the security of VoIP communication by identifying vulnerabilities in GUI-based SIP softphones. The provided state-based extension of the fuzzer framework fuzzolution allows for testing SIP states by sending SIP messages and GUI actions to control the softphone, e.g., initiate calls.

Multiple analyzers are integrated to automatically monitor the behavior of the SUT to detect vulnerabilities of VoIP softphone implementations. The responses of the SUT are analyzed and the port of the application is monitored. Additionally, the framework uses the log files of the softphones and observes the GUI behavior of the SUT, which utilizes changes of window states to determine errors, e.g., newly opened dialogs or closed windows. The CPU and memory usage of the process are monitored to detect extensive usage after sending the test attacks. The implemented OCR analyzer could not find an error because the displayed text was not recognized correctly. However, the screenshots taken by the OCR analyzer are helpful for further manual checks.

Messages are sent directly from the fuzzer to the SUT for some test scenarios and for other scenarios a proxy is used to send the messages. Instead of using Asterisk as done for previous tests we implemented a special and robust SIP registrar and proxy for the tests. Asterisk filtered some of the invalid messages. The current used implementation does not filter messages.

With the implemented fuzzer framework, two open source SIP-based softphones were tested and various security vulnerabilities could be identified, e.g., DoS, memory corruption, improper validation of array index, use of uninitialized variables and a kind of XSS. The amount of vulnerabilities found in previous tests showed that further extensive security tests with additional scenarios and variations are required for the softphone applications. The current executed tests also uncovered many security problems of the implementations. As further work the possible SIP states should be configured automatically to get the whole SIP state space for tests.

The previous version of the fuzzer framework produced many false-positive results. In the current version the accuracy of the fuzzer test result was increased by improving the analyzers on the one hand and on the other hand by using a combination of the single analyzer results. To reduce required time for manual analysis many additional functionality has been implemented, e.g., screenshot capturing or grouping of test results based on the analyzer outputs.

The test execution was done on Linux systems. For future versions it is required to implement GUI interaction and observation for additional operating systems.

With the presented approach various vulnerabilities of SIP-based softphone implementations could be identified. The results show that GUI interaction and observation is required to automatically test for security vulnerabilities of softphone applications efficiently.

References

[1] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser, and T. Grechenig, "Automated security test approach for sip-based voip softphones," in *The Second International Conference on Advances in System Testing and Validation Lifecycle, August 2010, Nice, France*. IEEE Computer Society Press, Aug. 2010.

[2] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "RFC 3261: SIP - Session Initiation Protocol."

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 6–6.

[5] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2008, pp. 206–215.

[6] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009.*, May 2009, pp. 474–484.

[7] C. Schanes, "fuzzolution fuzzer framework," 2011. [Online]. Available: http://security.inso.tuwien.ac.at/esse-projects/fuzzolution/

[8] H. H. Thompson, "Why security testing is hard," *IEEE Security & Privacy Magazine*, vol. 1, no. 4, pp. 83–86, 2003.

[9] E. Y. Chen and M. Itoh, "Scalable detection of SIP fuzzing attacks," in *Second International Conference on Emerging Security Information, Systems and Technologies, 2008. SECURWARE '08.*, Aug. 2008, pp. 114–119.

[10] P. Oehlert, "Violating assumptions with fuzzing," *Security & Privacy, IEEE*, vol. 3, no. 2, pp. 58–62, Mar./Apr. 2005.

[11] T. Zourzouvillys and E. Rescorla, "An introduction to standards-based voip: Sip, rtp, and friends," *Internet Computing, IEEE*, vol. 14, no. 2, pp. 69 –73, 2010.

[12] D. Endler and M. Collier, *Hacking Exposed VoIP: Voice Over IP Security Secrets & Solutions*. New York, NY, USA: McGraw-Hill, Inc., 2007.

[13] A. D. Keromytis, "Voice-over-ip security: Research and practice," *Security Privacy, IEEE*, vol. 8, no. 2, pp. 76 –78, 2010.

[14] W. Werapun, A. A. El Kalam, B. Paillassa, and J. Fasson, "Solution analysis for sip security threats," in *Multimedia Computing and Systems, 2009. ICMCS '09. International Conference on*, 2009, pp. 174 –180.

[15] C. Wieser, M. Laakso, and H. Schulzrinne, "Security testing of SIP implementations," 2003.

[16] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*. New York, NY, USA: ACM, 2007, pp. 47–56.

[17] U. Oulu, "PROTOS: Security testing of protcol implementations." [Online]. Available: https://www.ee.oulu.fi/research/ouspg/Protos,2010-02-14

[18] C. Wieser, M. Laakso, and H. Schulzrinne, "Sip robustness testing for large-scale use," in *SOQUA/TECOS*, 2004, pp. 165–178.

[19] D. Aitel, "The advantages of block-based protocol analysis for security testing," Tech. Rep., 2002.

[20] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a stateful network protocol fuzzer," pp. 343 – 358, 2006.

[21] H. J. Abdelnur, R. State, and O. Festor, "Advanced fuzzing in the VoIP space," *Journal in Computer Virology*, vol. 6, no. 1, pp. 57–64, 2010.

[22] T. Alrahem, A. Chen, N. DiGiussepe, J. Gee, S.-P. Hsiao, S. Mattox, T. Park, A. Tam, and I. G. Harris, "INTERSTATE: A stateful protocol fuzzer for SIP," 2007.

[23] I. G. Harris, T. Alrahem, A. Chen, N. DiGiussepe, J. Gee, S.-P. Hsiao, S. Mattox, T. Park, S. Selvaraj, A. Tam, and M. Carlsson, "Security testing of session initiation protocol implementations," *ISeCure, The ISC International Journal of Information Security*, vol. 1, no. 2, pp. 91–103, 2009.

[24] L. Feng and S. Zhuang, "Action-driven automation test framework for graphical user interface (GUI) software testing," *Autotestcon, 2007 IEEE*, pp. 22 –27, sept. 2007.

[25] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *RT '07: Proceedings of the 2nd international workshop on Random testing*. New York, NY, USA: ACM, 2007, pp. 1–1.

[26] J. Neystadt, "Automated penetration testing with white-box fuzzing," Feb. 2008. [Online]. Available: \url{http://msdn.microsoft.com/en-us/library/cc162782.aspx}

[27] Z. Xiaochun, Z. Bo, L. Juefeng, and G. Qiu, "A test automation solution on GUI functional test," *6th IEEE International Conference on Industrial Informatics, 2008. INDIN 2008.*, pp. 1413 –1418, july 2008.

[28] J. Bo, L. Xiang, and G. Xiaopeng, "Mobiletest: A tool supporting automatic black box test for software on smart mobile devices," in *Proceedings of the Second International Workshop on Automation of Software Test*, ser. AST '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 8–. [Online]. Available: http://dx.doi.org/10.1109/AST.2007.9

[29] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers, "Session Initiation Protocol (SIP) Basic Call Flow Examples," 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3665.txt

[30] M. Ronniger, F. Fankhauser, C. Schanes, and T. Grechenig, "A robust and flexible test environment for voip security tests," in *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, Nov. 2010, pp. 1–6.

[31] V. Lebedev, "Qutecom," website, 2011. [Online]. Available: http://www.qutecom.org/

[32] E. Ivov, "Sip communicator," website, 2011. [Online]. Available: http://www.sip-communicator.org/