Security Testing Over Encrypted Channels on the ARM Platform

Fatih Kilic
Chair for IT-Security
Technical University of Munich
Garching near Munich, Germany
e-mail: kilic@sec.in.tum.de

Benedikt Geßele
Department Secure Operating Systems
Fraunhofer AISEC
Garching near Munich, Germany
e-mail: benedikt.gessele@aisec.fraunhofer.de

Hasan Ibne Akram
Safety & Security Lab
Matrickz GmbH
Unterscheissheim near Munich, Germany
e-mail: hasan.akram@matrickz.de

***Abstract*—Security Testing has been applied for many years to detect vulnerabilities in applications. With the increasing demand for encryption to protect the confidentiality of network data, the requirements have changed. When proprietary, closed source software uses end-to-end encryption, security testing tools which are fuzzing the application layer over network with plaintext data will eventually fail. The Intrusion Detection Framework for Encrypted Network Data (iDeFEND) framework circumvents this problem without violating the security of the end-to-end encryption. Unfortunately, the framework cannot be used on the Advanced RISC Machines (ARM) platform, since it uses architecture depended features of x86. In this paper, we transfer iDeFEND to the ARM architecture and thereby, make it suitable for testing applications on embedded devices. In addition, we discuss the limitations of the current framework and improve it with novel methods to provide a more generic approach for security testing. We present a generic method for inspecting data on encrypted channels. Our approach does not require any knowledge of the structure of the wrapper function for receiving and decrypting like iDeFEND. Furthermore, we present a solution to test and inspect applications that are using packet queues. Finally, we evaluate our approach on popular mobile applications.**

Keywords–security testing; network security; reverse engineering; encrypted communication; embedded security.

I. Introduction

Nowadays, a wide variety of applications use encryption to protect their confidential data in network communications. Encrypting the network traffic prevents attackers from accessing sensitive data, but cannot stop them from exploiting security flaws in the implementation to achieve crashes, intrusion or code execution on the system. Security testing is responsible for detecting these vulnerabilities at an early stage. However, even powerful testing frameworks are blind when end-to-end encryption is applied and can only randomly generate or mutate packets. Additionally, the encryption layer makes it difficult for security testers to validate the remote program which increases the risk of missing faults. Solutions to this issue usually require a high amount of reverse engineering, since most of the target applications are closed source. Other solutions add an additional node to the encryption (e.g., a proxy server) and use it to access the plaintext data. This makes the communication more insecure. End-to-end encryption is designed to only terminate at the destination application to fulfil its required security. As a consequence, the plaintext can only be accessed by reverse engineering of the encryption algorithm and key, which is in general highly complex and time-consuming and thus, not feasible.

Another solution is presented by the generic framework iDeFEND [1]. The framework sustains the end-to-end encryption and leaves the communication channel untouched by extracting the plaintext data directly from process memory. It automates the reverse engineering process of applications by only relying on the detection and hooking of network and encryption functions. As a result, even closed source software can be handled at a much smaller effort. Although the framework has a generic design, it still has limitations. iDeFEND was implemented and evaluated for the x86 architecture, but nowadays most of the networking applications are running on mobile devices like smart phones or tablets whose processors are primarily designed by ARM. Since the framework uses hardware dependant features, its concept must be adapted to the specifics of the new platform.

Additionally, mobile applications tend to buffer network packets in a queue before sending them. This compensates bad connectivity, but results in a conflict with the current approach of iDeFEND. Furthermore, the framework relies on the presence of a specific wrapper function to inspect the received, unencrypted network data. In practice, this function can be more complex than expected by the framework and requires additional reverse engineering.

We overcome these shortcomings and extend the iDeFEND system. We provide a framework that allows to use common security testing tools for encrypted network applications. In summary, our contributions are the following.

- **Security testing over encrypted channels on ARM**
  We provide the same features of iDeFEND for ARM as it already does for x86. This means, we enable security testing on ARM devices when the target applications are communicating over an encrypted channel.
- **Improving iDeFEND to support applications with packet queues**
  We improve the current approach of iDeFEND with a new feature that makes it capable of handling applications with packet queues. Our new method allows to inject plaintext data into the packet queue and thus, into the encrypted communication channel.
- **Improving iDeFEND with a generic method for data inspection**
  We extend the concept of iDeFEND by a generic method for extracting received network data. We describe how this method enables the inspection of server responses without reverse engineering the function in detail.

The remainder of this paper is structured as follows. First, we present related work in Section II. In Section III, we summarize and describe the approach of the existing iDeFEND

framework. How the framework is used for security testing is explained in Section IV. In Section V, we present the limitations of the current concept and introduce our design improvements. In Section VI, we discuss the conceptual transfer of iDeFEND from x86 to ARM. The implementation of iDeFEND on ARM follows in Section VII. In Section VIII, we evaluate our framework and summarize the paper in Section IX.

II. Related Work

Many different fuzzing frameworks exist that facilitate the security testing of network communicating applications. Gascon et al. [2] present a fuzzing framework for proprietary network protocols which uses inference to create a generative model for message formats. Their approach relies on unencrypted network traffic, similar to many other smart automated model-based [3][4][5] and grammar-based [6][7] fuzzing techniques. Nowadays, there is also a vast amount of powerful commercial fuzzing and vulnerability scanning frameworks like Defensics [8], Nessus [9], beSTORM [10], Peach Fuzzer [11], honggfuzz [12] and american fuzzy lop [13] on the market available. They provide very complex and sophisticated algorithms to cover many different areas of fuzzing and vulnerability testing, but overall also lack proper support of encrypted network communications.

Biyani et al. [14] address this issue and present a solution by extending the SPIKE fuzzing framework to support encrypted protocols. They add a SSL wrapper to the existing plaintext fuzzer which allows to communicate with the target test application over an encrypted tunnel. This way, the fuzzer can inject its plaintext test data into the encrypted channel and test the target application for vulnerabilities. This approach, however, is limited to Secure Sockets Layer (SSL) encryptions which only represent a small part of proprietary software products. Another drawback is that their implementation is customized and only applicable for the open source fuzzer SPIKE. Tsankov et al. [15] introduce a different solution that allows a more generic fuzzing of encrypted protocols. Their approach is based on the knowledge of the encryption key and algorithm which is problematic from a security point of view.

As of yet, there is no good solution to testing of applications with encrypted network traffic. Our approach is different. We use the iDeFEND framework [1] to have a layer between test program and test framework. This additional layer makes the encryption transparent without violating the security of end-to-end encrypted communications. This way, we reduce the problem of testing encrypted protocols to the testing plaintext protocols and thus, enable the usage of many already existing testing tools.

III. Design of iDeFEND

In this section, we summarize the iDeFEND [1] framework and describe how the framework enables inspection and injection of plaintext data in encrypted communications. We also show why the approach is well suited for security testing.

Usually, applications implement encrypted communication with the help of two wrapper functions. One takes plaintext data, encrypts it and sends it over the network. This function is labelled EnCrypt & Send (CaS). The other one, Receive & DeCrypt (RaD), is responsible for the reversed process. It receives ciphertext data from the network and decrypts it afterwards. Together, these functions form the transition between plaintext and encrypted network data in our target applications. The iDeFEND framework uses this property to get access to the unencrypted network data by detecting and hooking both wrapper functions. This way, the application itself serves as an abstraction of the encryption implementation and allows us to inspect the plaintext communication without knowing the encryption algorithm, key or even source code of the application.

Controlling the wrapper functions empowers us to inspect, intercept, modify and inject new plaintext messages into the encrypted channel. For security testing, especially fuzzing, the tester primarily wants to send test data to the remote application and thus, heavily relies on the injection of packets. Since the CaS wrapper function takes a plaintext data pointer as argument, encrypts it and sends it over the network, test data can be injected by passing its pointer to the CaS. This can be realized in two different ways. Either active by code injection to the target process and calling CaS directly or passive by hooking calls to CaS inside the application (e.g., with a debugger) and replacing the input plaintext pointer with a pointer to the test data. In both scenarious, the test data is sent to the remote application, the response is extracted at the RaD and the test case can be evaluated.

The functionality of iDeFEND is logically split into three modules: a detector, a collector and a monitor module. The detector module is responsible for locating the wrapper functions in memory. Afterwards, the collector module hooks the located wrapper functions and passes the plaintext data to the monitor module. The monitor module simply is an interface for external programs. The detector module is a debugger that is specifically geared towards the automated reverse engineering of the wrapper functions. In general, applications with encrypted network traffic implement the functions *crypt*, *send* and *receive*. Send and receive are public library functions of the operating system and thus, getting their addresses is simple. The crypt function, depending on the underlying algorithm, can either be one or two functions. In case it is part of a library, getting the addresses is simple. They can be extracted by looking at the export table. In case it is not, the paper for interactive function identification [16] introduces an approach that facilitates the identification. By definition, the wrapper functions successively call the pairs encrypt and send, and receive and decrypt, respectively. iDeFEND uses this property of CaS and RaD to identify the wrapper functions through backtracking with a debugger. The backtracking is realized with breakpoints on send, receive, enrypt and decrypt. When the debugger notices a break on one of the function pairs, it can determine the wrapper functions from the call stack. Sometimes data is only encrypted for internal purposes and never sent over the network. In order to filter those cases, iDeFEND compares the data pointers between the function calls and validates the data flow. Data for network communication is detected, for instance, if the output pointer of the encryption matches the input pointer of the send. Otherwise, the calls of encrypt and send were independent and did not originate from the wrapper function, but from an internal encryption.

The collector module hooks the detected wrapper functions and extracts the network data. It is either part of the debugger or a module that is injected into the target application.

- **Collector Module as a Debugger**

Extracting the plain text with a debugger is simply achieved by inserting breakpoints on the wrapper functions CaS and RaD and extracting the data from their function arguments and return values, respectively. Since the debugging procedure is comparably slow, the target application is slowed down to a certain degree.

- **Collector Module as an Injected Module**
  A faster solution is to directly place code in the target application with a module injection. An assembly hook that is placed at the function prologue of the wrapper functions CaS and RaD redirects the execution to the injected code. The hook consists of a machine instruction like a jump or a call that substitutes the first few bytes of the function prologue and a function stub that is executed by the jump. The extracted plain text data then is passed via Inter Process Communication (IPC) to the monitor module.

IV. Security Testing with iDeFEND

In this section, we present an use case of the iDeFEND framework and explain how it enables security testing of encrypted network applications.

The iDeFEND framework is designed to support security testing of proprietary, closed source software. This type of testing is referred to as black box testing, since we examine the functionality of the programs under test without knowing details on the development, program internals or implementation. Even though the program is a blackbox, security analysts still can use powerful fuzzing tools to test for commonly known vulnerabilities. They can, for example, test a server against blind format string attacks [17]. In this scenario, a security analyst sends strings to the server application and afterwards validates the response and thereby, the outcome of the test case. For applications that use an encrypted communication channel, this approach of security testing inevitably fails. Since no information about implementation and design of the target application are available, also the internals of the encryption are unknown. As a result, there are only two possible responses of the target application to plaintext test messages from the security analyst. Either the test message does not fulfil the specification of the protocol and thus, the decryption fails and the test data is rejected. Or the decryption handles the test data, but changes it arbitrarily and is interpreted differently to the intentions of the tester. In both cases testing fails. Figure 1 illustrates this scenario with the orange arrow representing the test string data. The diverging arrow heads symbolize the misinterpreted test data after decryption that does not trigger the intended functionality any more.

Figure 1. Security testing of encrypted communications.

If the security analyst wants to test the server application as intended, he can use the iDeFEND framework. Using the framework for testing circumvents the issue of encryption. It provides an interface for the security analyst to the client application and thus, access to the encrypted channel. This way, the security analyst can pass the plaintext test data to the framework interface which uses the client application to encrypt and send the data. The sent data then is decrypted correctly at the server application and eventually triggers the intended functionality. Figure 1 shows the flow of the plaintext test data with the dashed, green arrow. The security analyst passes the data to iDeFEND which injects it into the encrypted channel. The test data enters the server application and is decrypted correctly.

V. Improvements of iDeFEND

In this section, we discuss the limitations of the current iDeFEND approach for software testing and present our improvements. We put focus on the conceptual weaknesses of the framework and separately address the transfer to ARM in the following section VI.

Currently, iDeFEND implements the identification of the wrapper functions with backtracking. Therefore, the call stacks at successive calls to the logic function pairs are intersected. Knowing, for example, that wrapper CaS is responsible for calling encrypt and send, means that the call stacks of encrypt and send must have an intersection at the wrapper function. This approach introduces a weakness. The wrapper functions can only be detected when they successively call encrypt and send. For applications that use a message queue in network communication, this assumption is never met.

Additionally, iDeFEND defined the RaD wrapper function to return the decrypted plaintext packet. It inspects the plaintext data by hooking the function at its return instruction. This requires detailed knowledge about the structure of RaD and obviously the presence of a RaD.

In the following subsections we propose solutions to those two problems.

A. *Test Data Injection into Packet Queues*

Applications that use a packet queue construct the packet, encrypt it and then append it to the queue. At any other point in the program the encrypted packet is taken from the queue and sent over the network. As a result, the call graphs of encrypt and send do not intersect at the CaS, because there is no CaS any more. This introduces a weakness of the iDeFEND framework. Without the detection or presence of the wrapper function, the framework cannot inspect, intercept or inject data into the communication. This means, for applications that use packet queues it is not possible to use iDeFEND for security testing. We addressed this issue and analysed the program structure of such applications and came up with a solution. Even though the applications do not implement a CaS function, they still have a function that takes the plaintext data, encrypts it and appends it to the queue. This function can be used in the same manner as the CaS to inspect, intercept and inject data to the communication. The only difference is that the sending is delayed in time, which is irrelevant to our scenario of testing. Figure 2 illustrates the control flow graph for this new function type EnCrypt & EnQueue (CaQ). Identifying the address of this wrapper function requires a new approach. Usually, programs implement protocols that construct different packets for many different purposes. This means that for each packet the wrapper function is called from a different calling

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-564-7

context, but their call stacks always intersect at the CaQ. For this reason, our solution to the issue of identifying the CaQ function is to record all call stacks at encrypt and intersect them to find the wrapper function. In order to validate network traffic in this scenario, it is also necessary to use a different procedure to the previous. Since the data is copied to the queue, the pointers at send and encrypt vary. We handle this problem by not saving the pointer itself, but the whole buffer. At the validation of the data flow we simply compare the contents.

The CaQ function can be identified as soon as at least two call stacks from different calling contexts are collected. The intersection of the collected call stacks identifies the wrapper.

This proposed method extends iDeFEND to support applications that implement packet queues.

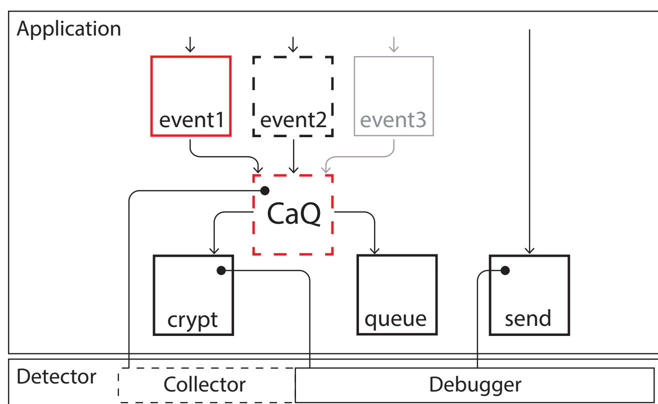

Figure 2. Control Flow Graph (CFG) for wrapper function CaQ.

B. Generic Approach for Data Inspection

The second problem of iDeFEND is that the current approach assumes the existence of a specially structured RaD function which in general is not the case. The RaD is assumed to return the decrypted plaintext data. iDeFEND hooks the RaD at the return and extracts the plaintext data. However, many applications do not implement this type of wrapper function. In general, the receiving wrapper function is a loop that never returns. It calls receive and passes the data to a parsing unit that finally decrypts the data. Furthermore, without knowing the structure of the RaD, the current iDeFEND cannot inspect the plaintext data. The correct offset and the information about the correct register or data pointer have to be known at this point.

We analysed this issue and came up with generic solution. Our approach is based on the assumption that data that is received over network is always decrypted at any later point in the program. Therefore, we store all incoming data at the function receive and wait for it to be decrypted. When the decryption function is accessing the data, we can extract the plaintext after the decryption has completed. This way, we do not need the presence of wrapper functions or knowledge about the function structure, but only require the presence of the the basic functions decrypt and receive. Additionally, our improved approach does not even rely on frame pointers.

Similar to the original approach, we also break on receive and decrypt. However, we identify data that is received from the network not by comparing the pointers of data, but by comparing the content of input and output buffer between receive and decrypt. The idea is the same as it was for validating data that is going to be send over the network for the CaQ. When the decrypt function returns and we validated that the encrypted data was received from the network previously, we extract the plaintext data from the returned buffer. The extracted data then can be passed to the tester for inspection.

With this method we extended iDeFEND to allow the inspection of unencrypted server responses, even though the application does not implement a wrapper function and use frame pointers.

VI. Transfer to the ARM Architecture

In this section, we discuss the transfer of iDeFEND to the ARM platform. We present the key differences between x86 and ARM with respect to debugging with hardware breakpoints, data extraction at function calls, call stack reconstruction from the stack and hooking of functions on machine code level.

A. Using Hardware Breakpoints for Debugging

Both architectures x86 and ARM implement both hardware and software breakpoints. Hardware breakpoints offer a better performance, do not require modification of the executable code and thus, are less obvious to detect. This makes them perfectly suited for implementing the detector module of iDeFEND.

In general, only a few hardware breakpoints are available per processor, but this is no limitation, since the specification of x86 offers up to four and ARM up to 16 hardware breakpoints. Implementing the detector requires at most four breakpoints. On x86, each debug register represents a breakpoint and holds the target address. A shared control register holds flags to enable, disable and configure each breakpoint. On ARM, hardware breakpoints consist of two registers: a control and a value register [18]. The value register stores the address of the breakpoint and the control register contains breakpoint options that allow, for example, to link different breakpoints, enable or disable them, specify the privilege and exception level the breakpoint debug event is generated on.

B. Extracting Data from Procedure Calls

The collector has to extract data from the function parameters on breaks. Since we break on function prologues, which means on the first instruction of the routine, we can access the passed parameters as specified by the underlying calling convention.

ARM, in contrast to x86, specified its own procedure call standard [19]. On ARM, the first four parameters are always passed in the first four registers `R0` to `R3`. Every additional parameter is pushed to the stack. Since the *Stack Pointer* register always holds the address of the top of the stack, the arguments five and higher can be accessed with help of the stack pointer register and the argument offset.

In case the output buffer is passed through the return value of a function, the *Link Register* can be used to access it. The *Link Register* is dedicated to hold the return address of the current function. The return value is passed through register *R0*.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-564-7

TABLE I. PRESENCE OF STACK POINTERS WITH DIFFERENT COMPILER SETTINGS

GCC Flag	Optimization				Offset to next frame pointer (FP)
	O0	O1	O2	O3	
no flags	✓				FP - 4
-mapcs-frame	✓	✓	✓	✓	FP - 12
-fno-omit-frame-pointer	✓	✓	✓	✓	FP - 4
-mapcs-frame fno-omit-frame-pointer	✓	✓	✓	✓	FP - 12

C. Call Stack Reconstruction

In order to identify the wrapper functions CaS and RaD, we want to intersect the call stacks and therefore, have to reconstruct them from the program stack. In a program every function call pushes a frame to the stack and pops it on return. The call stack can be reconstructed by unwinding the stack frame by frame. On ARM, unwinding the stack is complex. In general, the architecture does not provide a dedicated frame pointer register for the address of the first frame. Depending on the optimization level of the underlying compiler, frame pointers might not even be present on the stack. This is problematic, since it is highly complex to reconstruct stack frames without having frame pointers, as it requires a sophisticated analysis of the stack. Table I illustrates the effects of different settings on the generation of stack frames for the GCC compiler. The flags *mapcs-frame* and *fno-omit-frame-pointer* force the compiler to preserve stack pointers throughout all optimization levels. The only difference is that the pointer offsets vary. Without them, the compiler only generates stack pointers for optimization level O0, which means no optimization. In the default case, without any particular flag specified, frames are properly build by the compiler.

D. Hooking Functions

Injecting the collector module into the target process requires a redirection of the control flow from the original code to the injected module. Therefore, a hook is placed in the executable code at the prologue of the target function. Generally speaking, this means substituting the first bytes of the function prologue with a branch instruction. The replaced code must be backed up and executed later on, before jumping back to the original function.

On ARM, instructions have a fixed length of four bytes, which makes substitution of instructions simple. However, multiple types of prologues exist. This is problematic when the first instruction is program counter dependant and thus, cannot be moved. This happens on ARM, for example, when compilers use constant pools. Otherwise, when the instruction is independent of the program counter, the instruction can be moved and a hook is possible.

The actual branch can be implemented with a memory load that allows to target the full 32 bit address space. Since it modifies the program counter directly, the hook consists of only one instruction plus memory space that is holding the target address. Since compilers use multiple bytes of padding between two procedures in memory, this padding is a suitable location to place the address.

VII. IMPLEMENTATION

We implemented the improved iDeFEND framework on an ARM device that is running a Linux operating system. We chose Linux, as most of the target ARM devices like smart phones, tablets or embedded boards are either running Linux or Android, which is also based on the Linux Kernel. We used a Raspberry Pi 2 embedded board that is equipped with a 900MHz quad core ARM Cortex-A7 processor and 1GB RAM. It was running a Linux distribution Raspbian 4.1.13-v7 as operating system. For the sake of efficiency, portability to Windows and independence of other programs and their implementations, we decided to write our prototype as a stand alone C program.

The implementation consists of two parts. First we present the detector module, followed by the implementation of the collector module.

A. The Detector Module - a Debugger based on ptrace

We implemented the detector on top of the *ptrace* debugging API by setting four hardware breakpoints for each of the target functions.

1) Finding addresses of Send and Receive: Since Linux maps the whole binary object to memory, the virtual addresses of send and receive can be calculated by adding the offsets in the binary to the base address of the module process space. We retrieve the base address and path to the binary on disk from the directory */proc*. We then use the utility *nm* to find the offsets inside the binary.

2) Detecting Successive Calls to Function Pairs and Locating the Wrapper Functions: In order to locate the wrapper functions, we identify successive calls by extracting the function arguments at encrypt and at receive, and see if they match the input pointers at send and decrypt. For the special CaQ case, we copy the whole buffer instead of only pointers. We track the data per thread, together with a time stamp. A 15 seconds time out prevents internal encryptions to stay infinitely long in memory. We implemented stack unwinding for applications compiled with *-mapcs-frame*. As described in Table I, each frame pointer minus twelve then points to the previous pointer. After reconstruction, we intersect two call stacks by searching for the first frame that appears in both call stacks.

B. The Collector Module - Speed Up with Module Injection

We implemented the collector in both variants debugger and injected module. For injection, we implemented a call to *dlopen* that uses the dynamic loader of Linux to load objects at runtime. Finally, we placed the hooks at the wrapper functions and detoured the execution to the injected module.

VIII. EVALUATION

We have evaluated our improved iDeFEND framework for five applications. Beside the required criterion of encrypted network communication, we wanted to have at least one messenger, one file transfer and one secure shell application. These types implement different network protocols which handle text messages, binary files and customized commands. Furthermore, we wanted to have at least one test application that is single-threaded, multi-threaded, uses the console for user interaction and implements an own Graphical User Interface (GUI). In order to have ground-truth information of the wrapper functions, we used open source applications. Table II gives an overview of the selected applications telegram-cli (v1.4.1), uTox (v0.7.1), PLINK (v0.67), PSFTP (v0.67)

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-564-7

TABLE II. DESCRIPTION OF THE OPEN SOURCE TEST APPLICATIONS THAT RUN ON A RASPBERRY PI 2

Name	Type	Data Category	UI	Threading
telegram-cli	Messenger	Text	Console	Multi
uTox	Messenger	Text	GUI	Multi
PLINK	Secure Shell	Commands	Console	Single
PSFTP	File Transfer	Files	Console	Single
PSCP	File Transfer	Files	Console	Single

TABLE III. EVALUATED APPLICATIONS

Name	Send	Receive	Wrapper Type
telegram-cli	Write	Read	CaQ
uTox	SendTo	RecvFrom	CaS
PLINK	Send	Recv	CaS
PSFTP	Send	Recv	CaS
PSCP	Send	Recv	CaS

and PSCP (v0.67). The second column states the type of the application. The third column shows the type of data that is primarily transferred by the protocol. The last two columns indicate whether the application implements a GUI or is multi or single threaded, respectively.

Table III summarizes the results of our evaluation. The first column contains the names of the applications. The columns send and receive state the system library functions the application used to communicate over the network. The column Wrapper-Type states whether a CaS or CaQ function is implemented. Briefly summarized, we were able to inspect, intercept and inject data for all five applications. Except for Telegram, all applications implement the CaS function. Telegram implements a message queue and therefore, uses the CaQ. With the improved approach we were able to detect it and use it for packet injection. We were also able to use code injection and hooking of the wrapper functions on all five applications.

IX. CONCLUSION

With the rising demand for confidentiality and thus, encryption in consumer level and commercial software, security testing faces new challenges. Currently, existing testing tools only have poor or no support at all for encrypted network communications. That is precisely the reason why we proposed a generic solution to this issue by using the iDeFEND framework. The framework makes the encryption transparent and thereby, does not violate the security of end-to-end encryption. Since iDeFEND cannot be used on the ARM platform and nowadays many network applications are from the mobile sector and thus, use ARM processors, we transferred it to the this architecture. Additionally, we pointed out the limitations of the current framework and introduced improvements to it. Our novel methods provide a more generic approach for security testing. We introduced a method that allows to inject test data into network applications that use message queues. Our solution detects and hooks the function that is responsible for encrypting and enqueuing packets.

Furthermore, we introduced a generic method to inspect the incoming unencrypted network data. Our method does not rely on the presence of a receive and decrypt wrapper function or even frame pointers.

With the extended iDeFEND framework we provide an interface to the encrypted channel of an application that allows already existing testing tools to work as intended, also on the ARM platform. Our improved framework decouples the testing of software from the actual encryption.

REFERENCES

[1] F. Kilic and C. Eckert, “idefend: Intrusion detection framework for encrypted network data,” in Proceedings of the 14th International Conference on Cryptology and Network Security (CANS 2015), ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9476, pp. 111–118.

[2] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. Cham: Springer International Publishing, 2015, pp. 330–347.

[3] T. Kitagawa, M. Hanaoka, and K. Kono, “Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols,” in Computers and Communications (ISCC), 2010 IEEE Symposium on, June 2010, pp. 202–208.

[4] G. Banks et al., SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 343–358.

[5] S. Gorbunov and A. Rosenbloom, “Autofuzz: Automated network protocol fuzzing framework,” IJCSNS, vol. 10, no. 8, 2010, p. 239.

[6] D. Yang, Y. Zhang, and Q. Liu, “Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs,” in 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, June 2012, pp. 1070–1076.

[7] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’08. New York, NY, USA: ACM, 2008, pp. 206–215.

[8] Codenomicon. Defensics. [Online]. Available: www.codenomicon.com/defensics/ 2016.04.26

[9] T. N. Security. Nessus. [Online]. Available: www.tenable.com/de/nessus 2016.04.26

[10] B. Security. bestorm software security testing tool. [Online]. Available: http://www.beyondsecurity.com/bestorm.html 2016.04.26

[11] Peach fuzzer. [Online]. Available: http://www.peachfuzzer.com/ 2016.04.26

[12] honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. [Online]. Available: https://github.com/google/honggfuzz 2016.04.26

[13] M. Zalewski. American fuzzy lop: a security-oriented fuzzer. [Online]. Available: http://lcamtuf.coredump.cx/afl/ 2016.04.26

[14] A. Biyani et al., “Extension of spike for encrypted protocol fuzzing,” in 2011 Third International Conference on Multimedia Information Networking and Security, Nov 2011, pp. 343–347.

[15] P. Tsankov, M. T. Dashti, and D. Basin, “Secfuzz: Fuzz-testing security protocols,” in Automation of Software Test (AST), 2012 7th International Workshop on, June 2012, pp. 1–7.

[16] F. Kilic, H. Laner, and C. Eckert, “Interactive function identification decreasing the effort of reverse engineering,” in Proceedings of the 11th International Conference on Information Security and Cryptology (Inscrypt 2015). Springer International Publishing, 2016, pp. 468–487.

[17] F. Kilic, T. Kittel, and C. Eckert, “Blind format string attacks,” in Proceedings of the 10th International Conference on Security and Privacy in Communication Networks (SecureComm 2014), ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing, 2015, vol. 153, pp. 301–314.

[18] ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile. ARM Limited, Jun. 2016.

[19] Procedure Call Standard for the ARM Architecture. ARM Limited, Nov. 2015, document Version: ARM IHI 0042F, current through ABI release 2.1.