Experience report: How to extract security protocols' specifications from C libraries

Itzel Vazquez Sandoval and Gabriele Lenzini
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg, Luxembourg
Email: {itzel.vazquezsandoval,gabriele.lenzini}@uni.lu

***Abstract*—Often, analysts have to face a challenging situation when formally verifying the implementation of a security protocol: they need to build a model of the protocol from only poorly or not documented code, and with little or no help from the developers to better understand it. Security protocols implementations frequently use services provided by libraries coded in the C programming language; automatic tools for code-level reverse engineering offer good support to comprehend the behavior of code in object-oriented languages but are ineffective to deal with libraries in C. Here we propose a systematic, yet human-dependent approach, which combines the capabilities of state-of-the-art tools in order to help the analyst to retrieve, step by step, the security protocol specifications from a library in C. Those specifications can then be used to create the formal model needed to carry out the analysis.**

I. Introduction

To protect a system, cryptographic protocols need to be not only carefully designed but also rigorously implemented and verified for security. Certification standards for Information Technology Security such as the "Common Criteria for Information Technology Security Evaluation"[1] award the highest certification levels (EAL7, EAL7+) to companies that incorporate *formal methods* to verify the compliance of their products with expected security guarantees [1]. An ample set of approaches are available today for this task, also supported by automated tools: taken as input a formal model of a protocol, i.e. a clear and unambiguous description expressed in a formal language (e.g. *Applied π-calculus* for the tool *Proverif* [2]), they rely on logic-based techniques (e.g. model checking and theorem proving) to find flaws in the protocol against specific adversary models.

To create a protocol's formal model, the analyst requires to clearly understand the protocol's structure and logic, and to know the crypto primitives used, the parties involved and the messages exchanged between them. Ideally, the companies should offer such insights, from the high level design to the underlying protocol's implementation. Still, given the diversity of software development methodologies, the availability of up-to-date documentation is not always guaranteed. In *agile methods*[2] for instance, documentation tasks are dictated by the needs and experience of each development team, following the principle: "Working Software over comprehensive documentation" [3]. Projects with minimal comments in the code as their sole documentation are thus not hard to be found. Appropriate documentation can be missing as well for protocols developed following a *secure-by-design* [4] approach since companies, pressed by deadlines, have little time to dedicate to write crystal clear documentation and comments.

The analyst therefore often remains with the problem of extracting the required information from lengthy software projects, navigating through which, is a daunting task.

Here, we discuss *how to obtain a concrete high-level representation of a security protocol whose core algorithms are services provided by a library, assuming that the only documentation available about the library is the source code itself implemented in C*. The work takes inspiration from a real experience: asked to analyze the security of a private-by-default communication protocol that uses the security services provided by a proprietary library implemented in C, we were given nothing more than such library.

II. Problem context

Assuming that a protocol has been correctly designed, its security also depends on its implementation which must be error-free. Instead of risking to implement their protocols in-house, companies follow the recommended practice to rely on trusted *libraries* (e.g. [5], [6]) that offer access through well-defined interfaces (API). Such cryptographic services are frequently implemented in C, which is a relatively low-level programming language with simple data structures. Since the code written by programmers is almost directly translated into machine code by compilers, programs in C tend to have very high runtime performance [7], a desired feature for cryptographic functions that involve mathematical computations inherently expensive for the CPU.

But what if an analyst has access only to those C libraries? How can (s)he retrieve a model apt to be formally analyzed? We posed to ourselves the same questions, and assuming to work under the following conditions: **(i)** unavailability or scarcity of documentation; **(ii)** unknown specifications of the protocol; **(iii)** inaccessibility to any application requesting the library's services to fully set up the protocol. Such situation appears frequently when working with third-party software libraries. But it also arises when analysts and library's developers collaborate: documenting and analyzing security may occur in parallel, hence, documents may not be available yet when the analyst need them.

[1]ISO/IEC 15408-1:2009: https://www.iso.org/standard/50341.html
[2]https://en.wikipedia.org/wiki/Agile_software_development

Reverse engineering software: To extract knowledge from source code, an analyst can resort to *reverse engineering (RE)*, i.e. the examination of a system to identify its components and their correlation, and to represent it at higher levels of abstraction [8]. The piece of software to be reversed delimits the techniques and tools that can be applied, and the kind of diagrams that can be retrieved.

In security protocols' analysis, RE frequently refers to reversing executable binaries or programs to source code, for instance through system-monitoring tools, disassemblers, debuggers and decompilers. Here we use RE in the context of *software at code level*, i.e., to go from source code to a conceptual model. Stable and good-performing tools exist nowadays to automate the creation of diagrams, mostly complying with the UML ISO standard for modelling systems.

Given that RE's prevailing motivation is to understand how a system works, tools usually consider as input a running application. This input does not always exist though; as previously discussed, security protocols are often encapsulated in libraries, which have no executable files.

Contribution: We searched for tools capable of generating interaction diagrams of services implemented in non-executable C code and concluded that no fully automatic tool is capable of executing such task. Thus, we present a *chart recapping capabilities* of relevant related state-of-the-art RE tools; then, we *propose a methodology* which integrates those tools with manual tasks for the analyst, aiming to offer a practical pathway to retrieve a high-level protocol description, concise enough to allow the derivation of a formal model for a security analysis. Since its steps are independent of the implementation's logic, our methodology is *generic enough* to be applied for obtaining a high-level model (e.g. UML diagram) of any C library. To the best of our knowledge, no work addresses this specific problem.

III. Study Case: SecEngine's protocol

Our research scenario concerns a cyber security solution offered by a company to protect by default the privacy and reliability of digital written communications. The core of the software product is encapsulated in a library that provides the main services to set up protocols for handshaking, trust levels, keys management and encryption; for simplicity, we will refer to this library as `SecEngine`. `SecEngine` is implemented in C (C99) and contains $\sim$70 files, from which the longest has approximately 2524 lines of code while the shortest contains around 40. The cryptographic primitives used in the services rely on verified third-party libraries implementing OpenPGP[3]. `SecEngine` is open source; however, the specifications of the implemented services are not published anywhere. The only documentation available refers to the application release.

Since the security of the whole product strongly depends on `SecEngine`, it is imperative for the company to ensure that such component is correctly implemented according to expected security goals. The research that we pursue regards the application of formal methods to assess the security level of `SecEngine`. But as previously discussed, we require to comprehend the protocols' logic in detail before even thinking about formal verification tasks. The experience that we report here addresses such problem: *how to obtain a high level abstraction (e.g. a diagram) of the services provided by* `SecEngine`. For this task, our industrial partner gave us the source code, as well as a .dll binary of the library.

Note that, although in general source code compilation is achievable, it is not always an easy task, especially when several third-party libraries or automatic code generation are involved. Since this step was already covered for us with the .dll, we found practical to use Windows for this work.

IV. Retrieving a protocol from a library in C

Aside from the company's study case, the approach we suggest is to reverse engineer the source code of C libraries in order to retrieve a model that describe the protocols sufficiently well to proceed with a formal analysis.

We first sought for an automatic tool fulfilling all the following requirements: performing RE of code without an executable file; supporting the language C; and being capable to generate any kind of interaction diagram (e.g., sequence, message). In searching such a tool, we browsed the Internet, IT blogs, and forums, we glanced through academic papers, and we conducted informal and unstructured interviews with experienced software developers and architects working in industry. We looked at both commercial and non-commercial tools, although for the last ones we only tested the demo versions, which in general provide full features for a limited period of time. Notwithstanding our search, we did not find a tool meeting all the constraints and we were led to design a streamline process where we combine different tools (Table I), and increasingly extract and gather pieces of information until we had enough knowledge to be able to build a model useful for the analysis. The process has five steps.

Step (1): Extract and list the API specification of the library
Input: Compiled Libraries; **Output**: The APIs

When a library implements and offers security services (*e.g.*, handshake and authentication procedures), programs wanting to use it need to know its API, i.e., what functionalities they offer. DLL Export Viewer[4] for *dll* files or the regular export for *jar* files are good tools for getting an insight into the services that the code provides and a hunch of their functionalities via some keywords in the names.

Once we learn the (names of the) functionalities that can be invoked, the next goal is to understand what exactly they do, what are their constituent steps and how the protocol's components calling them interact, all which can be captured by an *interaction diagram*. These diagrams are particularly suitable for the task as they show sequences of messages sent among components of a system (s.a. classes, files, hardware), including passed parameter values and events occurring in

[3]https://www.openpgp.org/

[4]http://www.nirsoft.net/utils/dll_export_viewer.html

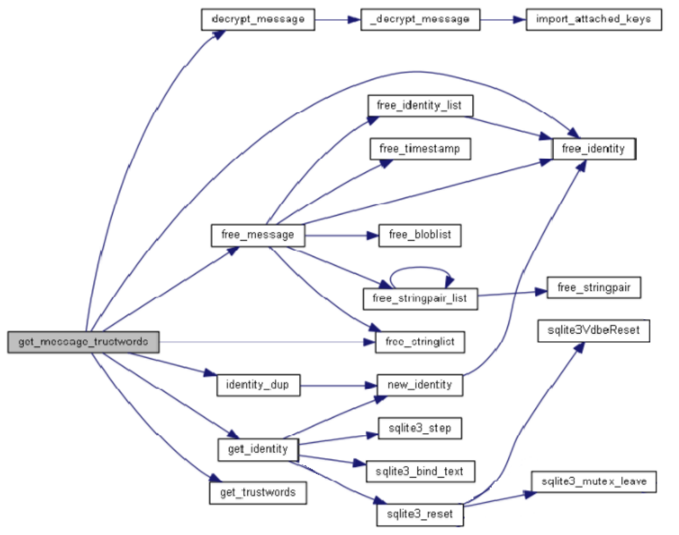


Fig. 1: Call graph for `get_message_trustwords` obtained by DOXYGEN.

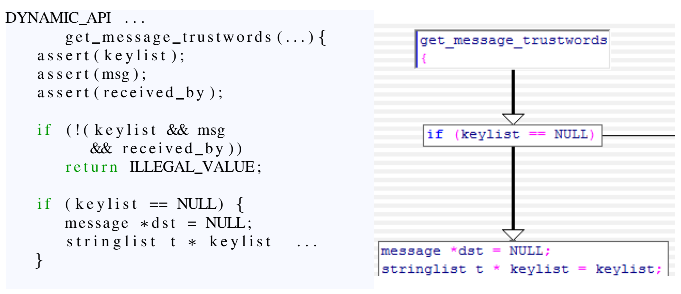
between [9]. Unfortunately, since a library does not run by itself, there is no easy way to gather this information. In fact, an intuitive solution would be to compile the libraries and create a simple application to request the services from the obtained binary; an existing tool could then be used to automatically get a model from such application. This approach is equivalent to implement a simple security protocol using the libraries. But the problem is circular: having no documentation, we do not know how to invoke the services and, moreover, we do not know yet what they do. In order to gain this knowledge we need to proceed with the next step.

Step (2): Create an overview of the protocol's architecture
Input: Library's source code; **Output**: graphs and diagrams

This step's goal is to create a global picture of the library's components and of the way in which they are related. Dependency graphs and collaboration diagrams accomplish this purpose. A convenient tool capable of automatically generating them is DOXYGEN, the standard tool to create documentation from annotated C++ sources, but it supports more programming languages, C among them[5]. The documentation is by default generated in HTML. A similar tool is OOVAIDE[6]. It creates classes and sequence diagrams, however, it was conceived for the Object Oriented approach and thus the functionality with C is incomplete. Auxiliary documents that can be created are UML *class diagrams* or the analogous *file diagrams* for C code [9]; they depict the relation among .c and .h files composing the implementation. *Call graphs* provide another useful visualization of the dependency among functions (Fig. 1).

Using the output from this step and keywords in the API functions' names (step 1), the analyst can identify and select the files and functions that contain code related to the logic of the protocol to be reversed, and discard code irrelevant for

[5]http://www.stack.nl/~dimitri/doxygen
[6]http://oovaide.sourceforge.net/



```
DYNAMIC_API  ...
        get_message_trustwords(...){
    assert(keylist);
    assert(msg);
    assert(received_by);

    if (!(keylist && msg
          && received_by))
        return  ILLEGAL_VALUE;

    if (keylist == NULL) {
        message *dst = NULL;
        stringlist t * keylist    ...
    }
```

```
get_message_trustwords
{
if (keylist == NULL)
message *dst = NULL;
stringlist t * keylist = keylist;
```

Fig. 2: Left: extract of `SecEngine`'s function `get_message_trustwords`, intended to retrieve words for authentication between two peers. Right: flowchart after removal of instructions related to parameters checking.

the protocol's description (*e.g.*, files implementing system's IO operations or interaction with databases).

Step (3): Retrieve algorithms of the relevant functions
Input: Library's source code; **Output**: Flowcharts

Call graphs from the previous step provide already an idea of a function's complexity in terms of the number of sub-functions invoked, but the order in which those sub-functions are called is missing. The goal here is to retrieve the algorithm that an identified relevant function follows. CODE VISUAL TO FLOWCHART (CVF)[7] converts procedural code into flowcharts and works with both, executable and non-executable declarations. The flowchart provides an easier way to follow the workflow of an algorithm, but since all the instructions are mapped into the diagram, it still needs to be refined by extracting only those instructions directly concerning the main purpose of the algorithm. C programs in particular contain a lot of low-level instructions, for instance to handle memory and pointers; code related to exceptions handling is also irrelevant in the normal flow of a protocol. After performing this step, the analyst has a flowchart with instructions related only to the protocol's logic or containing important assignments (Fig. 2).

Step (4): Identify relevant calls and group instructions
Input: Flowchart/Code; **Output**: Flowchart/Code

At the end of step (3) we have a sketch of the algorithm performed by a function but it is still too detailed. The objective in this step is to abstract the instructions into a higher-level conceptual model; the representation is still a flowchart/code but instead of direct programming instructions, the content will be in terms of tasks. For that purpose, we introduce two rules to be performed, which are similar to the *extract* and *inline* refactoring methods, used principally to reorganize code for better reuse and readability [10]: (1) Detect instructions linked to a single more general task and replace them all with a newly defined method with a meaningful name related to the purpose of the task; (2) For each function call: review the implementation details and if they are relevant

[7]http://fatesoft.com/s2f/

```
char *source1 = id1->fpr;
char *source2 = id2->fpr;

int source1_len = strlen(source1);
int source2_len = strlen(source2);
int max_len;

*words = NULL;
*wsize = 0;

max_len = (source1_len > source2_len?source1_len:source2_len);

char* XORed_fpr = (char*)(calloc(1,max_len + 1));
*(XORed_fpr + max_len) = '\0';
char* result_curr = XORed_fpr + max_len - 1;
char* source1_curr = source1 + source1_len - 1;
char* source2_curr = source2 + source2_len - 1;

while (source1 <= source1_curr && source2 <= source2_curr) {
  ...
  *result_curr = xor_hex;
  result_curr--; source1_curr--; source2_curr--;
}
...
status = trustwords(session, XORed_fpr, lang, &the_words);
```

After step (4)

```
XORed_fpr = combineFingerprints(id1, id2)
status = trustwords(session, XORed_fpr, lang, &the_words);
```

Fig. 3: Application of the abstraction rules. 1) goups the code above into the new function `combineFingerprints`. 2) keeps the call to `trustwords`.

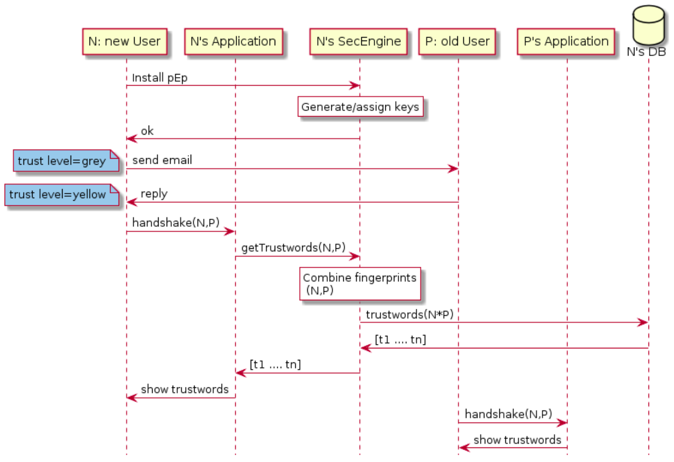


Fig. 4: Extract of the final protocol as a MSC created with PlantUML.

to define the protocol, mark the function (*e.g.*, by an '*'); otherwise, leave the call as a method.

These actions might be applied recursively to create flowcharts for the functions marked in 2 and so on. It is the analyst's duty to determine when an adequate level of abstraction has been reached.

Step 4 requires navigation through the files and the code itself, thus, any IDE would be in principle enough. We worked with VISUAL STUDIO 2015[8]; it is worth mention its functionality to view the call hierarchy of a function, since observing the references to and from a function helps to determine whether it has a principal or an auxiliary role.

Step (5): Create a diagram of the Protocol
Input: Flowcharts; **Output**: Message Sequence Chart

At this point, the analyst has a very deep understanding of the examined services' behavior, therefore, in the last step (s)he interprets the previous flowcharts to create a protocol's representation in the appropriate notation. There are many accepted notations; here we adopted Message Sequence Charts (MSC), defined in SDL[9]. This step is completely manual since it requires the analyst not only to connect information from the flowcharts, but also to use any domain knowledge to come up with the complete scenario of the protocols, including the actors involved. The participants can be for instance derived from the input parameters of the functions; as well, calls to databases imply the existence of a repository, which needs to be considered as an actor in the protocol model. The analyst's domain knowledge can come from different sources: a very general notion about the intention of the protocol can be learned in documents regarding final products using the library; often implementations include test files, which can provide examples of functions calls; user manuals can also contain hints about the components of the protocol and their interaction.

The modelling tool to be used is at choice of the analyst; we consider though more efficient the use of tools that automatically generate diagrams from scripts, given that the scripting languages' syntax tend to be simple and intuitive for programmers. A good performing tool in such category is PLANTUML[10] it supports all kinds of UML and some non-UML diagrams, and provides an online application as well as plugins to be integrated with many different IDEs. ZENUML[11] is another tool in this category; it has a syntax closer to C code an can be used online or as a Chrome extension, but is very limited in terms of the diagram it produces and also in the conversion to image formats.

V. EVALUATION AND REMARKS

According to our experience, this methodology is suitable to be applied in small and medium size projects since manual parts could become exhausting with very lengthy code. Based on the details in Sec. II, we consider `SecEngine` to be medium size.

We applied the methodology to reverse the protocols implemented in `SecEngine` and to generate the corresponding descriptions as MSC diagrams. The following outcomes report on our experience: Steps (1) and (2) can be executed relatively fast: one person/day was enough to generate the documents, scan them and identify the relevant functions. Step (3) requires attention since the analyst needs to concern about not discarding meaningful information; we invested 1 days per function in this task. Step (4) is the most time-consuming because here is where the analyst truly understands

[8]https://www.visualstudio.com/es/vs/
[9]The Specification and Description Language (SDL): http://www.sdl-forum.org/SDL/.
[10]http://plantuml.com/
[11]https://www.zenuml.com/

Step	Tool	Availability	Supported Languages	Step Output Diagrams	Output Format	O.S.
1	DLL Export Viewer	Free	dll	-	File Explorer	Windows
2	Doxygen	Free	C++, C, C#, PHP, Java, Python, ...	Call graphs, Dependency graphs, Inheritance diagrams	HTML, Latex	Windows, Linux, MacOS
	Oovaide	Free	C++, CLang related, Java	Zone, Class, Portion diagrams	SVG	Linux, Windows
2,3,4	Rational Rhapsody for Architect (RRA)	Free: 1 month, full functionality / Commercial	C, C++, Java, C#	Object model, Flowcharts	Several image formats	Linux, Windows
	Enterprise Architect	Free: 1 month, full functionality / Commercial	Ada, C, C++, Java, PHP, Python, ...	Several structural and behavioural diagrams	HTML, Several image formats	Windows, Mac/Linux (WINE)
3,4	Code Visual to Flowchart	Free for C,C++,C#: limited functionality / Commercial	C, C++, Java, PHP, PL/SQL, T-SQL, Perl, JavaScript ...	Flowcharts	PNG, only visual in the free version	Windows
-	Visual Paradigm	Free: 1 month, full functionality / Commercial	Java, C++	Class, Sequence, Communication, Component, Package, Object, Interaction Overview diagrams	Several image formats	Windows, Linux, MacOS
	Architexa	Free / Prototype for C on request	Java, Prototype for C++, C	Layered, class and sequence diagrams	PNG, own format	OS free. Plugin for the Eclipse IDE

TABLE I: Summary of RE tools reviewed in our methodology. The last two tools, do output sequence diagrams but only when taking as input Object Oriented (OO) code, not C. Although they do not work with our requirements, we include them here as a reference for analysts working with OO languages.

a program; (s)he has to decipher the control flow statements, such as conditions and loops, and try to reconstruct the logic behind. Reconstructing the algorithm of a function containing around 500 lines of code took a PhD student inexperienced in the tools up to two weeks of work (thus, we believe that with practice and experience, this time can be considerably shortened). An appropriate selection of the auxiliary tool may have influence in these steps, given the differences in the accuracy and comprehension levels of the diagrams produced. We included two well-known commercial tools in this research: RATIONAL RHAPSODY FOR ARCHITECT (RRA) and ENTERPRISE ARCHITECT. In general, with C libraries they do a good job for class diagrams and flowcharts; RRA's flowcharts are slightly the most comprehensive from all the tools. For sequence diagrams though, both commercial tools rely on exploring execution traces from running applications, thus they are useless for automation in our case. The final transition from step (4) to step (5) was achieved by the student in one week. Certainly, these times are relative and depend on the complexity of the project's architecture; small files can contain very complex logic, requiring hence longer time to be understood. The time we have reported for the work is only meant to give a relative estimation to whoever wants to contemplate the possibility of applying this technique in a project.

The methodology is quite straightforward and uses automatic tools to perform the main tasks; still it unavoidably requires human interaction to connect the steps. And even if could be conceived as a manual RE assisted by tools, it presents clear advantages over purely manual analysis: at the end of each step, there is already a document concerning the source code and, in a situation where documentation is still missing, it supplies the analyst with means to promote discussion and knowledge exchange with the development team. Those intermediate diagrams as well help the analyst to contextualize faster his/her mental process after interruption periods during the analysis, in case that they occur. Another important advantage is that, even before starting the code inspection, the analyst has an organized and complete overview of the library's structure including the connexion among functions; this is beneficial to address the RE efforts in the adequate direction. The methodology exposes hints and techniques that could also be applied to optimize a manual RE process.

VI. RELATED TOOLS AND APPROACHES

The most general approach to automatically reconstruct protocols' specifications observes the code at runtime and then records sequences of executed actions; it aims to describe the protocol from an application perspective. The survey in [12] reports on tools following this approach. A close problem is treated in [11], where the authors work on reverse engineering code in C++, however, their algorithm relies on the number of objects instances and so is not directly applicable in our case. Another flourishing technique to infer the application-level protocol specifications relies on network traffic observation. It is aimed to network protocols (e.g., HTTP, RPC and CIFS/SMB) implemented in any layer of the OSI model[12]. The central idea is to automatically reverse engineer the protocol message formats of an application from its network trace. This method is especially useful when a file to execute the protocol is unavailable because it only requires a system implementing the protocol to be running, even if it is not running locally. The work in [13] surveys Automatic Protocol Reverse Engineering Tools using this approach. Those tools seem to be mostly academic; a commercial one is VISUALETHER PROTOCOL ANALYZER 7.0[13], which uses the output of Wireshark (a network protocol analyzer) to generate sequence and call-flow diagrams. Avalle et al. [14] survey state-of-the-art research aimed at automatically getting formal security proofs of models close to the source code of real protocol-logic implementations. They comment extensively on work that extracts models to further validate widely deployed existing protocol implementations written in C (rather than libraries as in our case), limited to a subset of the language. In particular, they conclude that approaches for model extraction cannot deal

[12]https://en.wikipedia.org/wiki/OSI_model
[13]https://www.eventhelix.com/VisualEther/

with arbitrary legacy code, but introduce some requirements on how the code should be written, for instance by adding annotated semantic information. A relevant technique is reported in [15]. It aims to automatically create a formal model from an implementation in C. To drive the model extraction process though, it requires input from the analyst which demands him/her to have knowledge about both, the C language and the high level description of the protocol. Finally, although weakly referenced, Wikipedia offers a quite complete list of available tools for automatic reverse engineering into UML[14].

VII. Conclusion

We presented a methodology to retrieve a concrete and clear security protocol's description from undocumented source code implemented in C and without any executable file. The methodology, developed in five steps, combines commercial and free state-of-the-art tools to assist in the otherwise manual code-level reverse engineering process.

This approach offers a solution to whom needs to get to know (for instance to run a formal analysis) about the structure, the logics, the messages exchanged and their formats of security protocols encapsulated in C libraries but has to work in the scarcity of high-level documentation and/or definitions of security protocols, besides the lack of fully automatic tools to generate such definitions (e.g. interaction diagrams). State-of-the-art tools able to produce this output aim the Object Oriented paradigm, but little exists that work on non-executable code in C language.

Security analysts performing formal verification of software components that implement algorithms for security protocols can potentially benefit with the proposed technique, since core security services are frequently encapsulated and provided by C libraries but such libraries are not so often properly documented. Ensuring that they have a proper security level is however essential not to jeopardize the protection of software using them. By applying the methodology, the analyst can not only retrieve the protocols' specifications indispensable for the analysis, but also become much more aware of the vulnerable points of the system and detect security properties to verify.

Even though the scenario to reverse engineer executable applications is very advanced and efficient, the current situation for non-executable implementations still requires a lot of manual effort. Although our reverse engineering process is useful to achieve the final diagrams, its efficiency would highly improve when interaction with people knowing about the implementation or an insight in any kind of functionality-related document are possible.

The methodology presented here is the outcome of a practical research, testing and combining RE tools in the hunt of a model that capture the knowledge needed to perform a formal analysis, when the sole source of information about the protocol is the code itself. Considering the big amount of software that exists for reverse engineering, this involved an exhausting and also long process trying all candidate tools in the hope to find one useful for our specific problem. Therefore, with this report we attempt as well to make security analysts aware of the current RE situation and of the capabilities and limitations of existent tools; we intend our experience to prevent them from investing time and effort in such survey, allowing them to proceed further with the formal analysis tasks.

Our interest focused only in finding a solution that led to the assessment of security in implemented and scarcely documented protocols: our methodology has been indeed applied in a real industrial product herein briefly presented as a case study. We did not attempt to merge what we proposed into a fully automatic tool: despite conceivable it was a goal beyond the scope of our research but such a toolkit would have, we believe, its audience. Taking over this task is a challenge that we leave to software engineers working in the development/improvement of RE tools.

[14]https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

Acknowledgment

We thank the *pEp Security SA/SnT* partnership project ”Security Analysis of Protocols for Privacy” for supporting this research.

References

[1] H. Garavel and S. Graf, “Formal methods for safe and secure computers systems,” Federal Office for Inf. Security, Tech. Rep., 2013.

[2] B. Blanchet, “A computationally sound automatic prover for cryptographic protocols,” in *Workshop on the link between formal and computational models*, Paris, France, Jun. 2005.

[3] K. Beck, J. Grenning, R. Martin, M. Beedle, J. Highsmith, S. Mellor, A. Van Bennekum, A. Hunt, Schwaber, Cockburn, Jeffries, Sutherland, Cunningham, Kern, Thomas, Fowler, and Marick, “Manifesto for agile software development,” 2001, accessed: 2018-01-11.

[4] C. Dougherty, K. Sayre, R. Seacord, D. Svoboda, and K. Togashi, “Secure design patterns,” SFW Engineering Inst., Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2009-TR-010, 2009.

[5] Tech-FAQ, “Cryptographic libraries,” http://www.tech-faq.com/cryptographic-libraries.html.

[6] Wikipedia, “Comparison of cryptography libraries,” https://en.wikipedia.org/wiki/Comparison_of_cryptography_libraries.

[7] E. Eilam, *Reversing: Secrets of Reverse Engineering*. New York, NY, USA: John Wiley & Sons, Inc., 2005.

[8] E. J. Chikofsky and J. H. Cross, “Reverse engineering and design recovery: a taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[9] B. P. Douglass, “Uml for the c programming language,” IBM, Tech. Rep., June 2009.

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[11] P. Tonella and R. Potrich, “Reverse engineering of the interaction diagrams from c++ code,” in *In International Conference on Software Maintenance*, 2003, pp. 159–168.

[12] N. Tiwari and L. Prasad, “Reverse engineering tools for simplifying programming environment through flowcharting,” *Int. Journal of Eng. Trends and Technology (IJETT)*, vol. 26, pp. 65–71, 8 2015.

[13] J. Narayan, S. Shukla, and T. Charles Clancy, “A survey of automatic protocol reverse engineering tools,” *ACM Computing Surveys*, vol. 48, pp. 1–26, 12 2015.

[14] M. Avalle, A. Pironti, and R. Sisto, “Formal verification of security protocol implementations: a survey,” *Formal Aspects of Computing*, pp. 1–25, 2012.

[15] M. Aizatulin, A. D. Gordon, and J. Jürjens, “Extracting and verifying cryptographic models from C protocol code by symbolic execution,” in *Proceedings of the 18th ACM Conf. on Computer and Communications Security*, ser. CCS '11. NY, USA: ACM, 2011, pp. 331–340.