Project no.: **PIRSES-GA-2011-295261**

Project full title: **Mobility between Europe and Argentina applying Logics to Systems**

Project Acronym: **MEALS**

Deliverable no.: 3.5 / 1

Title of Deliverable: Analysing Cryptographic Protocol LLVM Implementations

Contractual Date of Delivery to the CEC:	**1-Apr-2013**
Actual Date of Delivery to the CEC:	**02-Oct-2014**
Organisation name of lead contractor for this deliverable:	**INR**
Author(s):	**Ricardo Corin, Felipe Manzano, Tamara Rezk**
Participants(s):	**INR, UNC**
Work package contributing to the deliverable:	**WP3**
Nature:	**R+D**
Dissemination Level:	**Public**
Total number of pages:	**9**
Start date of project:	1 Oct. 2011 **Duration:** 48 month

Abstract:

We report on progress towards analysing low-level cryptographic protocol implementations. In order to analyse concurrent communicating processes that use cryptographic functions, we work on top of KLEE, a symbolic execution engine, and KLEENET, an extension originally developed for analysing asynchronous wireless sensor networks. We model both standard socket-based communications and a symbolic layer that represents concrete cryptographic functions as (invertible) black boxes that can be efficiently manipulated. We present a prototype of our approach, illustrate it with some examples, and discuss next milestones towards a complete analysis.

This project has received funding from the European Union Seventh Framework Programme (FP7 2007-2013) under Grant Agreement Nr. 295261.

Contents

1 Introduction

Despite a large amount of research devoted to formally analyze cryptographic protocols (see e.g. [4, 3, 2]), the thorough and automatic analysis of existing industrial protocol implementations (like e.g. OpenSSL) remains largely unexplored.

Recently, methods based on symbolic execution (developed initially by [7]) were shown to scale very well to real life non-protocol code. Here, the code is dynamically explored through all its branches looking for implementation bugs like memory manipulation errors. In order to avoid trying the entire (arbitrarily large) input space, program inputs are assumed to be symbolic variables that remain uninstantiated (but become constrained) at execution time.

However, existing symbolic execution tools can't deal with code that uses cryptography: the search space blows up when exploring the insides of such functions, as they are specifically designed to avoid being invertible (e.g., hash or encryption operations), and hence the underlying constraint solver of the symbolic execution engine is unable to find suitable inputs in a reasonable time.

In order to cope with this problem, in recent previous work [5] we extend the symbolic execution engine KLEE by introducing "symbolic" functions that replace concrete ones, and prevent their exploration. In order to specify the behaviour of symbolic functions and to allow the execution to progress, symbolic functions are endowed with rewriting rules that detail abstractly their functional properties (e.g., that decryption inverts encryption).

In [5], we considered sequential code, matching the low-level virtual machine (LLVM) on which KLEE works. Thus, we were able to analyse only single-thread code, which is insufficient to analyse a whole protocol consisting typically of at least two communicating parties executing in parallel. In this work, we bridge the gap and develop a framework that is suitable for analysing such settings. We use an existing tool, KLEENET, built on top of KLEE. KLEENET is a framework originally developed for analysing asynchronous wireless sensor networks (WSN), where different nodes ("sensors") of the network run concurrently and become active or inactive (go to sleep) in different periods of execution. KLEENET allows to model specific network conditions of WSNs, like packet loss or duplication, and node reboots. The main difficulty solved by KLEENET is in scaling the symbolic execution with n nodes that communicate packets with potentially symbolic destinations (their so-called state mapping problem). In KLEENET, sensor nodes from the WSN compute for a while, then schedule themselves to wake up at some later time, and then go to sleep, in order to save on energy consumption. At a given time, KLEENET selects active nodes for execution, and advances time when all nodes are done for that turn. Nodes can write messages to other nodes' memory, and can signal them for waking up earlier (as an indication that they need to process a message).

In our case, we deal with processes (that is, cryptographic protocol participants) that are not designed nor rely on real-time computing. All the timing issues we need to model arise from communication (e.g., timeouts or blocked operations). Also, the ability of one node to write arbitrarily on another node's memory is overkill for our purposes; standard message-passing communication is enough. Thus, our first goal is to encapsulate KLEENET and implement an abstract socket library where messages are sent and received in standard socket-based fashion. A receive operation may block, or succeed and return with an indication of the number of bytes

read, possibly less than the length intended to be received. A send operation could succeed immediately or also potentially block, waiting for an operating system's sending buffer to be freed.

Hence, we add an extra process "System" that models an operating system that handles communication. Regular process do not talk to each other directly, but rather via System. Moreover, having a special System process brings us another advantage. Symbolic functions (like encrypted values) are symbolic terms that can be inverted according to rewrite rules. This is implemented via a table where symbolic function calls are table entries that record the different function parameters at the time of the call. Using this table, for instance, a destructor function (like decryption) is modelled as a table lookup, where the original plaintext is returned when the right key is provided. In the concurrent case, the table needs to be centralized and accessible from every protocol participant process, since for example a process can decrypt a value that another process had previously encrypted. Thus, we can conveniently store it in the System process, and provide system calls to serve queries from the regular processes.

We proceed as follows: in Section 2 we illustrate the need for concurrent symbolic execution with cryptography with a concrete example. Section 3 sketchs a concurrent semantics for LLVM. Section 4 presents our prototype, and Section 5 concludes the paper.

Related work. Previous research in this area focuses on extracting and verifying formal models from implementation code. This includes the work of Goubault [6], where protocols written in C are abstracted into horn-clauses. Another direction is taken by Bhargavan et al. [1], which develops an implementation of TLS coded in ML. In this case, the implementation is interoperable with other industrial implementations and can be analyzed by automatically extracting and verifying its corresponding formal model (using Proverif [4] and Cryptoverif [3]). Still, these tools are not applicable to legacy or binary code.

2 Symbolically Executing Concurrent Code

Consider the simple nonce exchange example of Figure 1 between A and B, who share a symmetric key K, and use it to encrypt the nonces. Function input() generates a fresh symbol. Functions send_to() and recv_from() are communication calls that talk to our System process, as decribed in the previous section. Function klee_assert(0) tells KLEE to stop execution and issue an error.

Here, A sends a (symbolic) nonce *na* encrypted as $encrypt(na, K)$ to B. Upon receiving and decrypting successfully the nonce (as *nar*), B answers with $encrypt(nar + 1, K)$, and then send its own (symbolic) nonce *nb* encrypted (as $encrypt(nb, K)$). When A receives, it decrypts and checks that $nar = na + 1$, then answers back with $nbr + 1$ encrypted as $encrypt(nbr + 1, K)$. Finally, B checks $nbr = nb + 1$.

At first glance, one could think of analysing the different protocol endpoints separately, using e.g. the standard KLEE machine. Messages received from the network are treated in the same way as regular inputs from the environment: they become fresh symbolic values that may take any arbitrary value.

However, when considering cryptography, this poses a problem. Consider for instance when A receives the second message $encrypt(nar + 1, K)$ as an array of symbolic bytes (for instance,

16 bytes for Rijndael), decrypts, and then checks whether $nar == na + 1$. Here, KLEE (more precisely, the underlying bitarray solver STP), would try to find 16 bytes that decrypt to $na + 1$, without knowing K, which is of course infeasible.

Another difficulty is that analysing a single process in isolation does not allow to specify and verify global properties that depend on all the running processes (think, for instance, of an authentication query modelled as a correspondence assertion, that needs events recorded by all participants).

On the other hand, if we execute both processes together in parallel and assume that cryptographic calls are symbolic functions that are not executed but rather stay as invertible terms, we are able to analyse the whole protocol.

The crucial point is that symbolic values and functions (corresponding to cryptographic calls) are communicated by KLEENET between different processes, thus binding symbolic values from different processes.

For instance, consider again when A decrypts the symbolic bytes corresponding to the second message $encrypt(nar+1, K)$ and proceeds to check whether $nar == na+1$. There is an interleaving trace where earlier in the execution the (symbolic function) $encrypt(na, K)$ was sent from A to B, and upon (symbolic) decryption the symbols na and nar where unified. This ensures the check $nar == na + 1$ succeeds, and we can continue exploring that branch without difficulties.

Using our current machinery we are able to explore all possible traces, reaching all possible combinations (A and B both print OK, or neither, or either one); this models a network that either delivers messages exactly as they were sent (making the checks to succeed), or that different messages were sent (out of order, duplicated, possibly modified enroute or completely forged), making the checks to fail. (In order to ensure an attacker can't provide $encrypt(ni, K)$ for $ni \neq na, nb$, we need to ensure K is secret. See Section 5 for more on this.)

3 Semantics

In [5] we provide detailed concrete and symbolic semantics for the sequential case. In the concrete case, the execution state is represented by a *context* c, defined as a tuple $c = \langle pc, \mathcal{M}, \mathcal{G}, fs \rangle$. Here, the program counter pc is an address (s.t. $\mathcal{M}(pc)$ holds the first byte of the next instruction to be executed). fs is a stack of function frames. Finally, $\mathcal{A}$ is a set of memory addresses reserved during the execution of this function.

In the symbolic case, contexts are endowed with a *constraint store* Σ, a sequence of symbolic conditions that are used to record the path conditions taken in each branch during execution. In practice, the solvability of Σ is decided by a bitarray SMT solver.

The semantic rules of [5] define an execution relation $c \xrightarrow{r} c'$ between contexts c and c', and LLVM operation r.

Concurrency. In order to model concurrency, we define a global concrete state as a tuple of execution states (i.e. contexts) $\langle c_1, \ldots, c_n \rangle$ plus a queue of messages Q, and then proceed with the straightforward rule concrete PAR:

```
void
a_node ()
{
    int na, nar, nbr, enar, enbr;
    na = input();
    send_to(B,encrypt(na,K));
    enar = recv_from(B);
    nar = decrypt(enar, K);
    if(nar == na+1){
         enbr = recv_from(B);
         nbr = decrypt(enbr, K);
         send_to(B,encrypt(nbr+1,K));
         printf ("A OK\n");
     }else{
        printf ("A NOT OK\n");
        klee_assert(0);
     }
}
```

```
void
b_node ()
{
    int nb, nar, nbr, enar, enbr;
    nb = input();
    enar = recv_from(A);
    nar = decrypt(enar, K);
    send_to(A,encrypt(nar+1,K));
    send_to(A,encrypt(nb,K));
    enbr = recv_from(A);
    nbr = decrypt(enbr, K);
    if(nbr == nb+1){
         printf ("B OK\n");
      }else{
         printf ("B NOT OK\n");
         klee_assert(0);
     }
}
```

Figure 1: Simple nonce exchange

$$\frac{c_i \xrightarrow{r} c_i' \qquad r \neq \text{send, recv}}{Q, \langle c_1, \ldots, c_i, \ldots, c_n \rangle \xrightarrow{i,r} Q, \langle c_1, \ldots, c_i', \ldots, c_n \rangle}\text{CPAR}$$

This rule says that whenever a single thread can evolve with an independent operation (not communication), then the whole global state can evolve as well.

We add the two communicacion operations: $send(i, M)$ and $recv(i, M)$, where M is a single byte:

$$\frac{c_i \xrightarrow{send(j,M)} c_i'}{Q, \langle c_1, \ldots, c_i, \ldots, c_n \rangle \xrightarrow{i,send(j,M)} Q.\langle i, j, M \rangle, \langle c_1, \ldots, c_i', \ldots, c_n \rangle}\text{SEND}$$

$$\frac{c_i \xrightarrow{x:=recv(j,M)} c_i'\{x \leftarrow M\} \qquad Q = Q'.\langle j, i, M \rangle.Q'' \qquad \text{no } recv(j, X) \text{ in } Q'}{Q, \langle c_1, \ldots, c_i, \ldots, c_n \rangle \xrightarrow{i,recv(j,M)} Q'.Q''\langle c_1, \ldots, c_i', \ldots, c_n \rangle}\text{RECV}$$

Here, notation $c\{x \leftarrow M\}$ means updating the binding with value M. These rules add messages to queue Q (when sending) or removes a message in the queue (when receiving), in which case the oldest (leftmost) message is provided.

We finally need to update the symbolic semantics. Instead of carrying out a single Σ per process, we carry out a single Σ shared and updatable by every single process in the following symbolic PAR rule:

$$\frac{\langle \Sigma, c_i \rangle \xrightarrow{r} \langle \Sigma', c_i' \rangle \qquad r \neq \text{send, recv}}{Q, \langle \Sigma, c_1, \ldots, c_i, \ldots, c_n \rangle \xrightarrow{i,r} Q, \langle \Sigma', c_1, \ldots, c_i', \ldots, c_n \rangle} \texttt{SPAR}$$

Notice that communication operations do not change the sigma, meaning that they do not introduce or remove constraints, so they remain the same as the concrete ones; (however they do potentially assign a symbolic term to a binding, thus modelling the exchange of symbolic values as implemented by KLEENET).

We conclude the section by stating an easy consequence of Lemma 1 of [5], which says that concurrent symbolic execution is sound, in the sense that it operationally corresponds to concrete concurrent executions:

Lemma 1 (Soundness of concurrent symbolic execution)**.** *Let tr be a symbolic trace (i.e., one obtained with the concurrent symbolic semantics of rules SPAR, SEND and RECV) with Σ_{tr} σ-satisfiable. Then trσ is a valid concurrent concrete trace.*

Here, Σ is σ satisfiable when substitution σ makes every constraint in Σ to hold.

4 Prototype

In order to test our ideas, we have developed a prototype in the form of a software layer on top of KLEENET. The System process described in the Introduction implements a small set of system calls:

send_to(X, M) sends data M to process X

recv_from(X) receives data from process X

wait() blocks until a signal is received

signal(X) sends a signal to process X

The underlying system call facility is implemented using the bare KLEENET interface, hiding the real-time passage to our processes. The parameter and return values exchanged between the regular processes and the System is implemented serializing and deserializing data values on a shared static buffer.

In order to implement symbolic functions, the System process maintains a table and provides pseudo system calls, for instance:

encrypt(n, K) pseudo encrypts n with key K;

decrypt(en, K) pseudo decrypts en with key K;

When called, *encrypt*(n, K) returns a fresh token and saves the parameters and return value relation in an internal table. Then *decrypts*(en, K) uses this table to recall the original plaintext for a given token, thus emulating the decryption procedure; note that this may potentially branch, as there could be several potential valid encryptions; also, notice that the plaintext n could very well be a purely symbolic value (like *na* in our example).

5 Conclusions and future work

Results so far are promising. This work only presents the machinery to *execute* concurrent processes that use cryptography, not to analyse security properties; the properties analysed are still standard KLEE, like memory manipulation errors or user-specific stated via *klee_assert*. As future work we are working towards developing security analysis for establishing secrecy and authentication:

- *Secrecy.* A *taint analysis* that tracks the flow of data in each trace. This is implemented by having different possible taint labels attached to data values (bindings and memory); then different LLVM operations "pass on" the taintings (this includes flow operations like conditionals). For instance, we assign a taint label "secret" to the key K of our example, and check that each sent message through the network is not tainted, entailing that K is indeed secret (this can be proved even against realistic, polynomial time attackers!).

- *Authentication.* A second analysis is about modelling the network attacker as he changes enroute messages. For instance, even though we can prove that an attacker doesn't know key K in the example, he could still perform replay attacks (e.g., replaying the first message in the example and causing $na == nb$) in order to perform authentication attacks. Although we do cover this simple replays, since we support fetching different similar encryptions from the shared table, we do not support arbitrary messages computed by an adversary (e.g., reencrypting or decrypting messages and computing new ones). So we need to model the knowledge of the attacker, and explore (symbolically) all the possible computed messages.

To conclude, we'd like to point out that although we aim at analysing cryptographic protocols, in principle our framework can be used to analyse any concurrent system efficiently, since replacing concrete functions by symbolic ones allows a modular analysis that would be impossible otherwise.

Bibliography

[1] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 445–459. IEEE Computer Society, 2013.

[2] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[3] Bruno Blanchet. Mechanizing game-based proofs of security protocols. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–25. IOS Press, 2012.

[4] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.

[5] Ricardo Corin and Felipe Andrés Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, volume 6542 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2011.

[6] Radhia Cousot, editor. *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

MEALS Partner Abbreviations

SAU: Saarland University, D

RWT: RWTH Aachen University, D

TUD: Technische Universität Dresden, D

INR: Institut National de Recherche en Informatique et en Automatique, FR

IMP: Imperial College of Science, Technology and Medicine, UK

ULEIC: University of Leicester, UK

TUE: Technische Universiteit Eindhoven, NL

UNC: Universidad Nacional de Córdoba, AR

UBA: Universidad de Buenos Aires, AR

UNR: Universidad Nacional de Río Cuarto, AR

ITBA: Instituto Técnológico Buenos Aires, AR