Prestudy: Formal verification of a one time pad crypto device

ADAM JOHANSSON

Master in Computer Science
Date: March 21, 2018
Supervisor: Roberto Guanciale
Examiner: Mads Dam
Swedish title: Förstudie: Formell verifikation av en engångskryptomaskin
School of Computer Science and Communication

Contents

Chapter 1

Introduction

1.1 Formal verification

All computer users have probably lost time or work to misbehaving software. Building bug-free programs is incredibly hard, if not impossible. Software fails even when the budget allows for rigorous testing, such as the Ariane 5 Satellite disaster, when a $1 billion dollar prototype exploded due to software failure. In other cases, like when the Therac-25 radiation therapy machine incorrectly over-radiated patients, it has lead to actual injury and death. This project aims to produce a *formally verified* crypto system employing the one time pad algorithm.

Formal verification is the act of mathematically proving properties of hardware or software. Compared to testing or static/dynamic checking of software, formal verification is strictly stronger allowing us to prove abscence of entire classes of bugs. At least if we make (admittedly rather large) assumptions regarding the correctness of non-verified parts of the system, such as hardware. Formal verification can be done at various levels of abstraction and the level of abstraction chosen demands trust in all non-proven subsystems below that level. If formal verification is employed at the program *language level* you would thus have to trust the compiler, maybe a runtime system, the operating system, hardware etc. The set of systems you rely on is sometimes referred to as the trusted computing base (TCB).

Minimizing the TCB leads to a higher assurance that the proofs produced are valid. Minimizing of the TCB can be achieved in two ways, either by proving properties at a lower abstraction level or relying on systems that already have been formally verified.

1.2 The crypto system

The task is to formally verify correctness and other security properties, defined in this prestudy as well as during the project, of a crypto device running on simple ARM hardware. The crypto algorithm chosen for this project is one time pad. The reasoning behind using one time pad is simplicity of implementation and its proven property of *perfect secrecy* [7]. Plaintext will be received over serial port and the key will be saved on the device along with the program. The resulting ciphertext from XOR-ing the plaintext and key will be sent on a separate serial port. The aim of building this prototype is to demonstrate the power of formal verification and, in the future, investigate whether formal verification can be used to convey to a potential customer which security properties hold for a given product.

Figure 1.1: Plaintext is sent from some sender to the crypto device which in turn sends the cipher to some receiver. There is no two-way communication and blocks marked red are considered to be in a safe network.

1.3 Serial protocol

In figure 1.2 protocol messages are defined. The same protocol is used whether data is plaintext or ciphertext and a message consists of two fields of 1 byte for index, refering a key block for encryption, and eight fields of 1 byte for data. 1.3 and 1.4 describes the format of index fields and data fields respectively. Indices are marked with a 1 in the most significant bit (MSB) and data fields with a 0 in the MSB. This allows us to have a very simple protocol but still be able to resynchronize if a single byte is lost.

Resynchronization can be done by first reading 10 bytes into a buffer, if the MSB of those 10 bytes do not match the pattern `1 1 0 0 0 0 0 0 0 0`, shift the buffer one byte and read one more byte until the pattern is matched. With 1 bit reserved in index fields the total number of bits possible for indexing is 14. With a key block size of 8 bytes, this means the maximum addressable key size is 128 KiB, which is sufficient for a simple prototype.

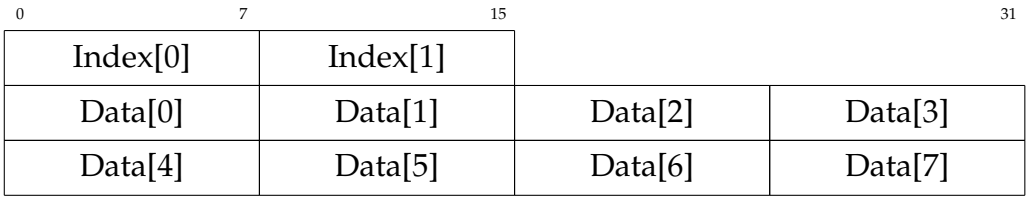

Figure 1.2: Protocol message with (exactly) two fields for key index and 8 fields for data.

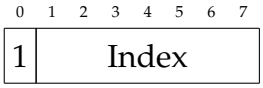

Figure 1.3: Bits of the index field. Indices always have the MSB set to 1.

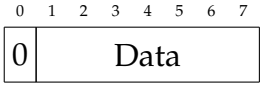

Figure 1.4: Bits of the data field. Data fields always have the MSB set to 0.

Let $0 \leq n < 8$ denote the data index and `i` be the 14 bit key index. Let `Key[i+n]` denote the key at index `i` with offset `n` bytes. The ciphertext will be produced by `Cipher[n] := Data[n]` $\oplus$ `Key[i + n]` and be serialized for transport by setting the MSB of each `Cipher[n]` to 0. The index and its field identifiers will be the same as the input message.

Chapter 2

Literature study

2.1 One time pad

One time pad is a symmetric cryptographic algorithm that is easy to implement and provably impossible to crack without the key. Its main drawback is that the pre-shared key must be at least as long as the plain text. The key also need to be truly random. The encryption scheme by Shannon [7] over a message of bits M of length l and a key of bits K is defined below. The operator $\oplus$ is the binary XOR operator.

$$M : \{0,1\}^l$$
$$K : \{0,1\}^l$$
$$Enc(k,m) = m \oplus k, m \in M, k \in K$$

The definition of *perfect secrecy* is, informally, that the probability of an adversary finding out the plaintext is equal with or without access to the ciphertext.

2.2 Hardware and models

The BBC microbit [1] is a widely available, well documented small computer with a 32-bit ARM Cortex M0 CPU with the ARMv6-M Thumb instruction set. It allows UART communication over USB and GPIO. Any device with sufficient amount of communication channels and an ARM processor would fit well for this project due to there being a good, well-tested model for HOL4.
The UART communication will be modelled in another thesis and that

model will be integrated at the end of this project. There is a model for the ARM processor and related peripherals such as memory. It is described in detail in [3] and models the fetch, encode, execute cycle with enough detail do have an instruction executed and its effects observed. It has been thoroughly tested on at least ARM Cortex M3 and ARM Cortex M8.

2.3 Security issues

2.3.1 Code injection and control flow injection

An attacker achieves code injection by leveraging bugs in applications, for example buffer overflows where input is written into the memory area after a buffer. Either that area already contains code that is supposed to be executed or the control flow of the application is manipulated to have the program execute the injected code. Control flow injection is when the attacker manipulates only the *control flow graph* (CFG) (or informally, manipulates the order of execution), to execute existing code in a manner not intended by the application developer.

An example of code injection is a *buffer overflow* attack where a bug in the program allows the attacker to overwrite code segments that lie after a buffer in memory. A common countermeasure is making all executable areas of the memory read-only which prevents code injection but not necessarily control flow injection.

2.3.2 Side channel attacks

A side channel is essentially an output or input of a system that occurs as a side effect of executing the system. Examples include, but are not limited to power consumption, electromagnetic radiation, inadvertent cache leaking, timing and acoustic properties of the system.

Side channel attacks can be mitigated, at least partially, by restricting physical access and providing special instruction sets which instructions power consumption and execution time is hard to analyze. On the software level, algorithms can add superfluous instructions to make all paths take a similar amount of time. The common goal of

all mitigations of side channels is to ensure that there are no observable difference between two executions where the non-secret inputs are identical but the secret inputs are different.

2.4 Correctness

Correctness is a property of programs and algorithms that holds if the algorithm is correct according to some form of specification. Hoare introduced an axiomatic system to reason about correctness. The main concept of that system is the Hoare-triple, denoted as $P\{Q\}R$ where P is a statement that holds before an execution, Q is the program to be executed and R is a statement that holds after the execution of the program Q. P is called the *precondition* and R is called the *postcondition*

The meaning of the notation $P\{Q\}R$ in the axiomatic system Hoare constructed relies on the assumption that the program Q terminates. This reliance is captured in the notion of *partial correctness*. That system was extended by Manna and Pnueli [4] to add the requirement of termination in order to prove *total correctness*, which is what this project mainly will be aiming for.

2.4.1 Composition

The rule of composition, as introduced by Hoare, says that if you have some triple $P_1Q_1R_1$ which is proven and you have another proven triple $P_2Q_2R_2$ where its precondition $P_2 = R_1$, both program sequences together are correct given the precondition P_1. Formally:

If $\vdash P_1\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R_2$ then $\vdash P_1\{Q_1; Q_2\}R_2$

2.5 Decompilation tool

There are two decompilation tools available that decompile machine code into HOL4 theorems. The first one was developed by Myreen, Gordon, and Slind [6] and was later improved by Fox [2]. The latter improved version is faster but more restrictive, leading to Anthony Fox's suggestion to go with the initial decompiler of Myreen.

The actual output of the decompiler is twofold, one or several theorems of Hoare triples $\{P\}\{Q\}\{R\}$ where P is a set of preconditions over state elements of the system, like the program counter, registers and their values as well as the memory state. Q is a list of machine code instructions and R is the postconditions defined analogously to the preconditions. The hoare triples represent a *total correctness* specification. The other output of the decompiler is a high level function f defined in HOL4 that describes the effects of the machine code and a precondition function f_{pre} that describes which side conditions must hold before executing f.

The $*$-operator is the *separating conjunction* operator borrowed from separation logic. The decompiler defines its $*$-operator over sets of system states in such a way that the operands are disjunct sets of system states.

Example:

{`(r0, r1, m)` **is** $(r0, r1, m) *$ pc $p * f_{pre}(r0, r1, m)$}

p: instr1 ... *p+x*: instrX

{`(r0, r1, m)` **is** $f(r0, r1, m) *$ pc $p + x$}

The first line in the example above are the preconditions. In informal terms, it says that we depend on two registers, `r0` and `r1`, the rest of the memory state is `m`. Furthermore the generated precondition $f_{pre}(r0, r1, m)$ must hold and the program counter pc must have the value *p*, pointing to the first instruction to be executed.

After executing all instructions, the postcondition of the generated function $f(r0, r1, m)$ must hold and the program counter points at the last instruction.

2.6 Similar work

A garbage collector was verified by Myreen [5], this verification included proofs in several layers of abstraction, using the interactive theorem prover HOL4 and the decompiler. The abstractions made higher

level proofs reusable for other garbage collectors while hiding the technical details of the theorems derived by Myreens decompiler. The theorems in the lowest level of abstraction are automatically generated by the decompiler from the machine code of the implementation. Theorems higher up in the hierarchy are proven manually in HOL4.

Smith and Dill [8] developed a system to automatically prove equality between implementations of block ciphers. The tool could also prove equivalence between a mathematical model and an implementation. The automatic proving was limited to algorithms that could be completely unrolled, that is, elimination of loops and recursive calls by repetition. The system used a formal model of the Java Virtual Machine (JVM) and used class files as input.

To the authors knowledge, no formal verification of a cryptographic system has been done at the machine code level. The automatic prover of Smith and Dill [8] requires trust in the JVM, the hardware and that the secret key is not leaked by other means.

Chapter 3

Discussion

3.1 Attacker Model

As described in figure 1.1 in the introduction, the crypto device is meant for use in a protected network. Thus, the attacker model does not allow for physical access to the crypto device. Since no other programs are running in the crypto device, all side channel attacks are considered out of scope for this project. The attacker may however somehow influence the sender to send a specific input so it needs to be safe from code injection and control flow injection attacks, such as buffer overflow or buffer overread. The attacker is considered to have infinite resources to attack the resulting ciphertext.

3.2 Security properties

Correctness of OTP system

This section introduces a high level specification of the crypto system. It will later be formalized in HOL along with an optional medium level specification and the low level specification derived by the decompiler. The task in HOL will be to show equivalence (for some notion of equivalence) between these different levels.

To start with, we define all input as some mapping between natural numbers and a list of bytes, $L_{Byte} = \mathbb{N} \rightarrow Byte$. The list of bytes can contain anything, both valid and invalid messages. In order to be correct, the crypto system needs to handle malformed input, which

it does by simply ignoring anything that isn't well formed input. To begin with, we want to check that the byte headers are correct and conform to the protocol. Remember that each message consisted of 10 bytes, the first two bytes is the sequence number and the other 8 bytes is data. The most significant bit (MSB) of the sequence number must always be 1 and the MSB of every data byte must be 0.

We define a function that marks each start byte as true of a message where the headers of all 10 bytes are correct. This is a function from some input $I \in L_{byte}$ to a list of boolean values $L_{Bool} = \mathbb{N} \to Bool$. $I(n)$ denotes accessing the n-th element of the list I, while brackets are used to access individual bits of a byte. For example, referencing the value in the first (MSB) in byte number 5 would be $I(5)[0]$.

$$\varphi_{start}(I) \colon L_{Byte} \to L_{Bool}$$

$$\varphi_{start}(I) = \lambda k. \left((I(k)[0], \ldots, I(k+9)[0]) = (1,1,0,0,0,0,0,0,0,0) \right)$$

So far we only have a function that marks the starting byte of a valid (in terms of byte headers) protocol message. In the next step, we use that function to find all the bytes that are part of a valid protocol message. Again, the input is defined by $I \in L_{Byte}$. The main idea here is to find the starting byte k' for every byte.

$$\varphi_{msg}(I) \colon L_{Byte} \to L_{Bool}$$

$$\varphi_{msg}(I) = \lambda k. \left(\exists k'. k - 10 < k' \leq k \wedge \varphi_{start}(I))[k'] \right)$$

Now that we know which bytes that are part of a protocol message with valid headers we can throw away invalid input. We make use of a standard filter function $\mathit{\Pi}$ that takes a list of elements and a list of booleans and returns the sub list of elements (in order) that are marked true at their corresponding index in the list of booleans. We call the resulting list V.

$$V = \mathit{\Pi}(I, \varphi_{msg}(I))$$

To have something more easy to reason about, we now introduce a type for messages as well as a function that generates such a list for us.

$$L_{msg} = N \to Byte^{10}$$

$$M\colon L_{Byte} \to L_{msg}$$

$$M(V) = \lambda k\left(V(10k); V(10k+1); \ldots; V(10k+9) \right)$$

Reading the sequence number from a message is rather straightforward. Take the 7 bits of the first byte of a message, bitshift 7 steps to the left and do bitwise OR ($|$) with the 7 bits of the second byte of a message.

$$seq(m) = (m[0][1:7] \cdot 2^7) | m[1][1:7]$$

Removing messages that have a sequence number less than some constant MAX.

$$\varphi_{MAX}(M) = \lambda k.\left(seq(V(k)) < MAX \right)$$

$$M_{MAX} = \Pi(M, \varphi_{MAX}(M))$$

We want to get rid of messages which sequence number is out of sequence, that is, if its sequence number is larger than the largest accepted so far.

$$\varphi_{seq} = \lambda k.\Bigg((k = 0) \vee \exists k'.(k' < k \wedge M_{seq}[k'] \wedge \forall t.(k' < t < k \wedge \neg M_{seq}[t]) \wedge seq(M_{MAX}[k']) < seq(M_{MAX}[k])) \Bigg)$$

This is the most complicated function so far, so lets break it down. The term $(k = 0)$ is our base case for this function. For the very first message out of M_{MAX}, its sequence number will trivially be the largest we've seen so far, thus it evaluates to *true*. The existential quantifier says that there must exist some $k' < k$ such that it its sequence number is valid, that is, $M_seq[k']$ holds. Additionally, we say that for all t in between that k' and out current k, $M_seq[t]$ does *not* hold. Thus $seq(MAX[k'])$ must be the largest accepted sequence number so far. If $seq(MAX[k])$ is strictly larger, the whole expression evaluates to *true*, otherwise, $seq(MAX[k])$ can't be part of a monotonically increasing sequence number and the expression evaluate to *false*.

Preparing for the actual encryption step, we need to be able to resolve a key. In one time pad we need a unique key for each message. These are indexed with the sequence number, pointing to a block of 8 bytes of a constant KEY.

$$key(seq) = KEY[seq * 8]$$

TODO the rest is much easier, but will leave it for now.

Code injection and control flow injection

Freedom from code injection and control flow injection follow from correctness, assume that a program P is correct, implying that for all inputs it produce a specific output. Code injection implies that an attacker may send some input and have the device execute arbitrary code to have the device behave outside of its specification. But that would mean correctness doesn't hold for program P. Thus, a correct program is free from code injection. A similar argument can be made about control flow injection.

Side channel attacks

While the attacker model does not allow the attacker physical access to the device nor the sender, timing and power measurements attacks are impossible. Electromagnetic or audiobased attacks can also be mitigated by physical constraints. It would however, in the future, be interesting to extend the attacker model and prove the crypto device free from sidechannel attacks.

Non-interference with UART driver

Proving the UART driver correct will be done after the UART is modelled in a separate thesis. The UART driver model will provide an interface that accepts a theorem of non-interference.

3.3 Choice of method

A stated goals of this project is to minimize the amount of software we trust in order to produce proofs that are relatively easy to understand and validate for a third party. By using Myreens decompiler [6], trusting the compiler is not necessary, the third party could potentially run the proof scripts themselves on a provided binary and see which security properties still hold for that binary. If this project were to use language level formal verification we would need to trust at least a compiler, which is contrary to the goal of minimizing the Trusted Computing Base and the implementation would be vulnerable to attacks targeting the compiler.

Ideally the theorems should be as human readable as possible, so that a third party can actually understand what is proven. This property can be achieved using any of the approaches described in the similar work section and is essentially done the same way, by abstracting away technical details and trying to provide higher level theorems. How many layers of abstraction that would be needed and if that level of abstraction can actually lead to a third party understanding the theorems without extensive knowledge of HOL4 remains to be investigated during the project.
Essentially, this leads to a choice of method similar to that of Myreen [5] in his proofs about the garbage collector.

3.4 Assumptions

One of the assumptions that have been identified at this point is that the key is truly random. That is, each possible key is chosen with the same probability p. This is very important for the one-time pad as described in Shannon [7], without this assumption, the proof of *total secrecy* would be impossible.

Additionally quite a lot of assumptions about the hardware and the model are made. For example, we assume that the ARM model is correct and correctly models an ARM Cortex M0 CPU, we assume that there are no hardware trojans in other peripheral hardware with DMA. Initially, we will also assume that the UART is correct, modelling UART and optionally providing a software implementation of UART will be the topic of another master thesis, the work of which will be merged with this project if possible.

3.5 Evaluation of results

The results of this project are the produced theorems and a working prototype. A proper evaluation of the theorems should include a section describing which assumptions that were made and what parts of the system we trust such that a reader can themselves get an insight into the strength of those theorems and the security properties they represent. The method can also be compared to other verification projects in cryptography by comparing assumptions.

Secondary result is a machinecheckable proof. Evaluation here is a bit less interesting, either the proof works or not.

Tertiary results are the proof scripts themselves, evaluation here is trickier since good properties could include reusability, understandability of code etc, which are all rather subjective properties.

The last result is that of the prototype. It will be considered successful if it is working and it is possible (with the help of simple helper programs) to send a message on one compute, have the prototype encrypt it and on a second computer be able to decrypt the message.

3.6 Trust and limitation of method

We trust the bootloader and the tools used, like the HOL4 kernel and that the ARM model is true to reality. Additionally, we have a high level of trust in the hardware. There is however no trust in the code of the implementation itself, in the compiler or any operating system.

3.7 Limitations of the prototype

The implementation will likely not be very useable in practice, there is no way to change or update the key, issues indexing plaintext and ciphertext limits the amount of total messages we can send. The resulting ciphertext isn't verified for integrity, that is, detecting modifications of the produced ciphertext is not possible without the user adding MAC or checksums in the message itself. OTP provides no authentication out of the box, so there is no guarantee that a sent message is actually sent by an authorized user unless implemented in the message.

3.8 The future

Relaxing the assumption that the attacker has no physical access would give a more reasonable attacker model and necessitate proving abscence of side channel attacks on power consumption or timing. Another venue to explore is to switch from UART, which is a bidirectional type of communication hardware to a proper data diode solution. One time pad, despite its feature of *perfect secrecy*, still is not a very useable encryption scheme due to the size of the key. Looking into other symmetric encryption algorithms would put this prototype closer to a working product. Additionally, adding checksums to the protocol will increase reliability.

3.9 Time plan

1. Milestone 1 – end of January
   (a) Prestudy complete
2. Milestone 2 – end of February
   (a) C implementation finished
   (b) Decompilation of implementation
   (c) Eventually, extensions to the decompiler to achieve above point
3. Milestone 4 - 15th February - 15th March

   (a) Protocol specification formalized in HOL

4. Milestone 5 – end of April

   (a) Code verification complete in HOL4
   (b) Correctness proof
   (c) Non-interference proofs
   (d) Necessary abstractions in the proofs

5. Milestone 6 – end of May

   (a) Delivery of Thesis report
   (b) Integration with UART project finished
   (c) Delivery of HOL scripts with UART integeration

Bibliography

[1] *BBC microbit website*. URL: http://microbit.org/ (visited on 09/12/2017).

[2] Anthony C. J. Fox. "Improved Tool Support for Machine-Code Decompilation in HOL4". In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. 2015, pp. 187–202.

[3] Anthony Fox and Magnus O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *Proceedings of the First International Conference on Interactive Theorem Proving*. ITP'10. Edinburgh, UK: Springer-Verlag, 2010, pp. 243–258. ISBN: 3-642-14051-3, 978-3-642-14051-8. DOI: 10.1007/978-3-642-14052-5_18. URL: http://dx.doi.org/10.1007/978-3-642-14052-5_18.

[4] Zohar Manna and Amir Pnueli. "Axiomatic approach to total correctness of programs". In: *Acta Informatica* 3.3 (Sept. 1974), pp. 243–263. ISSN: 1432-0525. DOI: 10.1007/BF00288637. URL: https://doi.org/10.1007/BF00288637.

[5] Magnus O. Myreen. "Reusable Verification of a Copying Collector". In: *Verified Software: Theories, Tools, Experiments: Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*. Ed. by Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–156. ISBN: 978-3-642-15057-9. DOI: 10.1007/978-3-642-15057-9_10. URL: https://doi.org/10.1007/978-3-642-15057-9_10.

[6] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. "Machine-code Verification for Multiple Architectures: An Application of Decompilation into Logic". In: *Proceedings of the 2008 International*

Conference on Formal Methods in Computer-Aided Design. FMCAD '08. 2008, 20:1–20:8.

[7] C. E. Shannon. "Communication Theory of Secrecy Systems*". In: *Bell System Technical Journal* 28.4 (1949), pp. 656–715. ISSN: 1538-7305.

[8] Eric Whitman Smith and David L. Dill. "Automatic Formal Verification of Block Cipher Implementations". In: *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. FMCAD '08. Portland, Oregon: IEEE Press, 2008, 6:1–6:7. ISBN: 978-1-4244-2735-2. URL: http://dl.acm.org/citation.cfm?id=1517424.1517430.