FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc thesis

Dominik Táborský

Lightweight Crypto in Reverse

Exploring lightweight cryptography in the context of reversible computing

Academic advisor: Michael Kirkedal Thomsen, Ken Friis Larsen

Submitted: March 19, 2018

Lightweight Crypto in Reverse

Exploring lightweight cryptography in the context of reversible computing

Dominik Táborský
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

March 19, 2018

MSc thesis

Author:	Dominik Táborský
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark
Title:	Lightweight Crypto in Reverse / Exploring lightweight cryptography in the context of reversible computing
Academic advisor:	Michael Kirkedal Thomsen, Ken Friis Larsen
Submitted:	March 19, 2018

Abstract

Abstract

Contents

Preface **1**

List of Abbreviations **3**

1 Introduction **5**
1.1 Terminology 5

2 Implementing cryptographic algorithms in reversible languages **6**
2.1 Implementations 6
2.2 Lack of state information leakage from data remanence 6
2.3 Duality of encryption/decryption 7
2.4 Other side channels 7

3 Practical programming experience in reversible languages **8**
3.1 How to 8
3.2 Already implemented improvements 8
3.3 Suggested further improvements 8

4 Conclusion **9**

Bibliography **11**

A The extra **13**

Preface

Preface and acknowledgements if you like to have this.

List of Abbreviations

....

Introduction

1

1.1 Terminology

directly reversible = an injective function

Implementing cryptographic algorithms in reversible languages 2

In this chapter we show the implementation of several cryptographic algorithms and discuss the advantages reversibility brings.

2.1 Implementations

2.2 Lack of state information leakage from data remanence

A major advantage over non-reversible languages is the required clean-up of intermediate results. For an example, consider the SIMON (NSA) algorithm. The algorithm takes a key and some plaintext as the input and is expected to produce ciphertext on the output. All other temporarily allocated data should be freed deallocated again. The Simon algorithm expands the key into a longer string, an *expanded key*, which is then used for the encryption itself. After the encryption is done, the intermediate expanded key is supposed to be wiped and deallocated. The issue with irreversible languages is that they cannot ensure the wiping of the expanded key. It is perfectly legal to only deallocate the memory, which can then be mapped by another process, making it readable.

Implementing the algorithm in a reversible language leads to either wiping the expanded key, or leaking it by intention, *i.e.* making it into a valid output. Unless the second option is favoured for some reason, we can uncall the key expansion, thus wiping the expanded key. Furthermore, in the extended version of Janus this becomes even more pronounced. Whatever is allocated as a local variable also has to be deallocated in the same scope. Therefore, by making the expanded key into a local variable, we are forced to wipe its state into a known value, presumably an array of zeroes.

We can observe this in the code. The expanded key is allocated at the beginning of the encryption and deallocated at the end. So far this is expected of any implementation, regardless of reversibility, thanks to common software engineering techniques like *RAII*. After other local variable declarations, the key expansion is performed, followed by the encryption itself. Just before deallocating local variables, we are forced to uncall the key expansion to get rid of the extra data. Reversibility forces us to provide the final value of all local variables, which, in the reverse execution, is also the initial value.

2.3 Duality of encryption/decryption

The second advantage is the fact decryption is the reverse of encryption. There is no reason to implement decryption functions, the encryption can be simply uncalled. The lack of decryption function has several implications:

- less time spent writing and debugging the code itself
- less time spent debugging caused by errorneous implementation of either encryption or decryption, leading to one being not the reverse of the other
- less code to maintain
- smaller library size

In any software project, these are valuable features.

2.4 Other side channels

is this section useful?

Unfortunately, due to the unavailability of real physical reversible circuits, we can only speculate about other side-channel vulnerabilities. One common side channel is the timing differences in execution that is input-dependent. One example are bit rotations. "Bit rotation by x bits" can be implemented by either directly rotating by x bits, or by x rotations by 1 bit. This difference made RC5 vulnerable to timing-based attack on some platforms [2]. Since bit rotations are directly reversible, there is no difference in comparison to irreversible implementations.

A directly irreversible operation is multiplication, assuming word-sized operations with modulo semantics. When implemented, it has to make use of the Bennett's trick. This implies that it will execute once to calculate the result, and once to wipe it. If it is timing-variable, the difference will only get more distinct in the execution time of the program.

I think this is right, but I'm tired.

Another side-channel vulnerability is variable power consumption. However, given the fact entropy stays the same throughout the computation, thus no energy is wasted, energy is used only by the supporting hardware. Ideally, this would imply no information can be gathered about the data within the system. Power consumption analysis is a viable attack against devices which the attacker has full physical control over, and it is an issue to be considered when designing, *e.g.*, smart cards. Still, without actual physical implementation, this can only be a speculation.

this does sound too good to be that simple and true

Practical programming experience in reversible languages 3

3.1 How to

bad section name

There are two general ways of implementing any algorithm in a reversible language:

1. Implementing it directly and manually reversing anything that produced intermediate data which we need to remove. This was usually the case with the actual encryption, as it tends to be straightforward in this case.
2. If any part of that algorithm is difficult to manually reverse, we can implement that part of the algorithm in its own procedure, while its output is stored in locally allocated variables by its caller. After the use of those values has passed, we uncall the algorithm, which resets the values in those variables, allowing for deallocating them. This is called the *Bennett's trick*.

3.2 Already implemented improvements

-

3.3 Suggested further improvements

- Constants that do not require explicit deallocation. Macros are fine.
- Functions with return values. These can be implemented using a hidden local variable with initialization to the function call and deinitialization by function uncall.

Conclusion

4

Discussion of results.

Bibliography

[1] Almeida, J. B., Barbosa, M., Barthe, G., and Dupressoir, F. Verifiable side-channel security of cryptographic implementations: Constant-time mee-cbc. In *Fast Software Encryption* (Berlin, Heidelberg, 2016), T. Peyrin, Ed., Springer Berlin Heidelberg, pp. 163–184.

[2] Handschuh, H., and Heys, H. M. A timing attack on rc5. In *Selected Areas in Cryptography* (Berlin, Heidelberg, 1999), S. Tavares and H. Meijer, Eds., Springer Berlin Heidelberg, pp. 306–318.

[3] Protzenko, J., Zinzindohoué, J.-K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., and Swamy, N. Verified Low-Level Programming Embedded in F*. *ArXiv e-prints* (Feb. 2017).

The extra

A