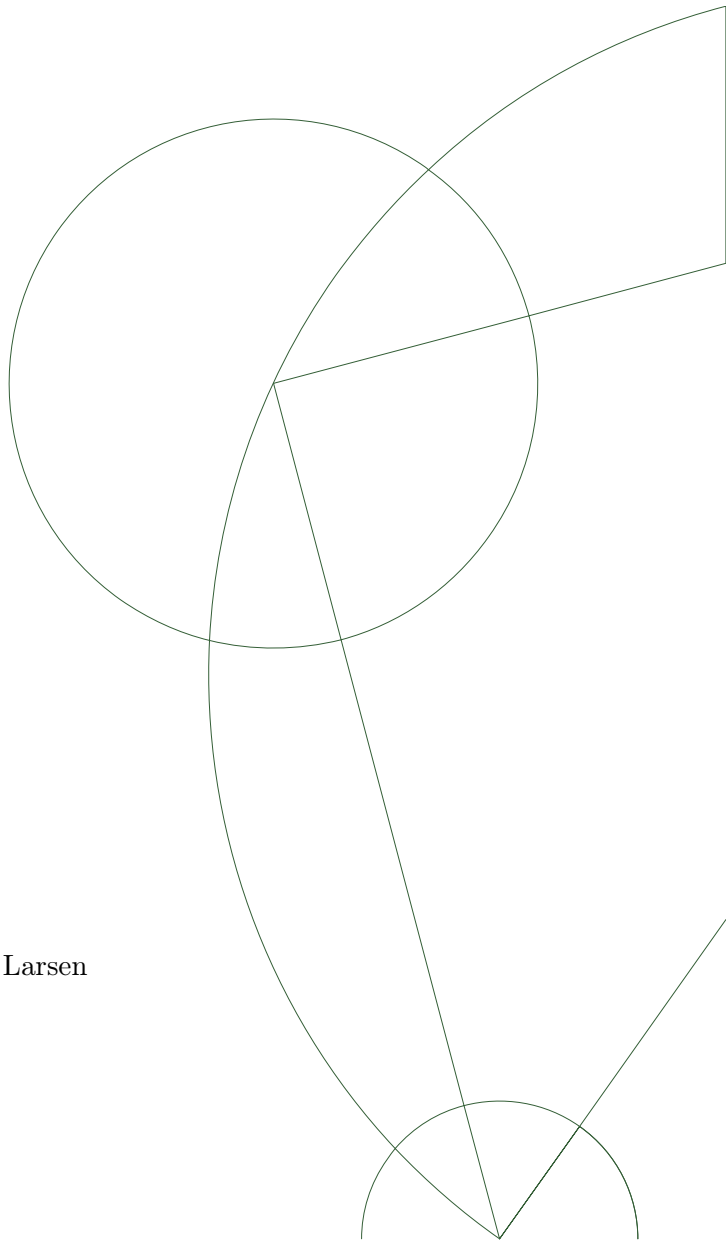
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

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: July 7, 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

July 7, 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:	July 7, 2018

Abstract

Abstract

Contents

Preface

Preface and acknowledgements if you like to have this.

List of Abbreviations

....

Introduction

1

Cryptography in general has been with us for decades now, yet only recently we finally figured out how to reliably code and verify an actual implementation with the F*/HACL work [**?**]. A similar approach takes the Vale project [**?**], designing a new language that utilizes either Dafny or F* for verification. These solutions verify not only the correctness of the code, but also prove the lack of state-based information leakage, and in the case of Vale, even timing-based leakage. These two are possibly the easiest side-channel vulnerabilities to exploit, especially given the fact that physical access to the computer is not necessary. Both of them can get difficult to control for since the compiler often provides no guarantees on the generated machine code. Similarly, programming languages do not provide semantics on low enough level to account for registers, caches and execution timing. Vale provides for these, which allows it to formally verify the cryptographic properties; on the other hand, it also makes the language difficult to learn and use due to the sheer breadth of details the programmer now has to consider, instead of leaving it to the compiler.

Our work aims for exploiting reversible computing to reason about state-based leakage. Informally, a reversible program has to follow a clearly defined path that transforms the input data to output data without losing information. Because of that lack of information loss, it can be described as an injective function. Encryption algorithms are bijective functions, implying they can be directly encoded in a reversible setting. In a formal description, we have

$$[\![enc]\!](k, p) = (k, c)\,.$$

The plaintext p is directly transformed to a ciphertext c, while key k remains the same. Note that the key cannot be omitted: either such the function would not be injective, thus irreversible, or the ciphertext would somehow contain the key, which would make the encryption useless by making the key public.

In Janus we enforce this scheme by offloading any other data into temporary local variables. These variables are declared and defined at the top of a function and undeclared at the bottom, with their final value specified. This final value is necessary for reversibility: without it, such reversed function would have undefined initial values. Secondly, because we know the final value of each variable, that value can be subtracted from the variable, thus clearing it. This implies that local data, not part of the input and output, will be cleared and thus no state-based leakage is possible. We demonstrate this by translating a Janus program to C, annotating it using special keywords for a LLVM plugin and compiling into machine code that can be manually verified to clean up used memory.

Furthermore, thanks to reversibility we implement only the encryption functions. The decryption is then obtained by reversing the encryption. This decreases time spent on development, debugging, testing and showing the duality of encryption and decryption functions.

Our conclusion is that reversibility is a handy tool for developing not only cryptographic code, but also other injective functions. It gives no guarantees on avoiding timing vulnerabilities like Vale does, however it comes at a much lower cost of development.

The reason for vulnerabilities based on the implementation and not the mathematical basis of the algorithm is usually performance, simplicity, inexperience and even language and/or compiler support for low-level semantics.

1.1 Overview of the chapters

Analysis

2

2.1 Symmetric and asymmetric cryptography

Cryptographic algorithms can be defined as functions that take two parameters, a key and some data, perform either encryption or decryption and return the key and the modified data. Normally the key in the result would be omitted, however it is useful to recognize the fact that the key is not altered in any way. This will become useful later on.

Symmetric cryptography differs from the asymmetric kind by its use of only a single key for both encryption and decryption. These are the classic algorithms like AES, RC5 and Chacha20. They also tend to be much more efficient performance-wise than their asymmetric siblings. This efficiency also lead to establishing the subarea of *lightweight cryptography*. Algorithms classified as *lightweight* are generally good candidates for use in *Internet of Things (IoT)* devices, which often have low computational performance, need to limit power usage, have strict memory requirements, or even may be required to react quickly. A special kind of symmetric cryptographic algorithms are stream ciphers, which only generate seemingly random output which is then combined with the input data (plaintext or ciphertext) using some self-inverting operation, typically XOR. Therefore stream ciphers only implement this "random" output and not individual encryption and decryption functions, since they are trivial and equivalent.

Asymmetric cryptography, also known as *Public Key Cryptography (PKI)*, uses one key for encryption (a *public key*) and another for decryption (a *private key*). Its security relies on some hard mathematical problems like integer factorization in the case of RSA. If that could somehow be performed as fast as the factor multiplication, it would render the whole algorithm broken. It does not achieve the same levels of performance . For that reason it is not considered lightweight. It can, however, be used for so-called hybrid encryption schemes, where PKI is used to exchange the secret key for some symmetric algorithm.

citation needed

Definition

Formally we use the following notation to describe a function *enc*that fits the description above:

$$enc_S(k, p) = (k, c)$$
$$dec_S(k, c) = (k, p)$$

This suffices to describe symmetric cryptographic algorithms including stream ciphers where $enc = dec$.

For asymmetric algorithms we have to differentiate between public and private keys:

$$enc_A(k_{\text{public}}, p) = (k_{\text{public}}, c)$$
$$dec_A(k_{\text{private}}, c) = (k_{\text{private}}, p)$$

However, notice one major difference between the encryption functions enc_S and enc_A: the resulting pair (k, c) of enc_S contains all necessary information to decrypt the ciphertext back using dec_S. That is not the case with enc_A: having access to (k_{public}, c) does not suffice for decryption, since we are missing the private key k_{private}. This is an important distinction that we will explore later in 2.3. In short, the implication is that the definition of enc_A is not as complete as enc_S.

2.2 Side-channel vulnerabilities in cryptographic code

The mathematical basis upon which cryptographic algorithms are built is only one half of the story when it comes to evaluating data security. The other half are their implementations, which have a long history of hidden vulnerabilities in various forms. These are called *side-channel vulnerabilities*, *side-channel information leakages* or just *side-channels* for short. These are classified into several groups depending on the type of the attack. The easiest (in terms of required prerequisites) are state-based and timing-based leakages, since they can be performed remotely and do not require any specialized hardware. Some other side-channels are power consumption and electromagnetic radiation analyses.

examples?

how to defend against them?

2.3 Reversibility and its implications

intro

Definition

Expressive power

Reversing asymmetric cryptography

When we formally defined functions enc_S and enc_A, we noticed the distinction in what information they give as a result. After looking at reversibility and seeing what it provides for symmetric cryptography, naturally it may seem like reversibility breaks asymmetric cryptography since an attacker knows both values, the public key and the ciphertext. We also know that reversibility does not increase expressive power of a language, it is just as powerful as a general Turing machine. So where is the problem?

The problem is that our definition was not complete. If the pair of the public key and the ciphertext contained all the information, then the algorithm would

not provide any security. The security is exactly in the fact that the attacker has to search for some hidden information that is not available to them. One way of looking at this is simply considering multiplication: multiplying 3 and 4 has only one result, but if we know the result is 12, we do not know what the factors were. If we knew one of them was 3, then we would have all the information and could just calculate the other one. With irreversible setting this is not a problem, we delete information all the time, but that is not an option with reversible languages.

The correct and complete definition of enc_A would then be:

$$enc_A(k_{\text{public}}, p) = (k_{\text{public}}, p, c)$$

Of course the plaintext is not transmitted or shared, only the ciphertext is. This definition avoids information loss and can be directly reversed by simply uncalculating the ciphertext. In the case of the RSA algorithm, the public key has two components which are combined with the plaintext input in a loss-ful way. How this operation works is well-understood, out-of-scope of this thesis and the details are not important for us here.

define "uncalculating"

Data transformation

Bennett's tricks

Implementations

Reversible languages

Reversible hardware

2.4 Goals

Design

3

+ use translation to other language for verification (e.g. Vale)

Implementation

4

Evaluation

5

5.1 design

5.2 implementations

5.3 experiments/testing

Reflections

6

Evaluation of our solution in a broader context, usability in practice and viability of real-world implementation

Related Work

7

obvious

Conclusion

8

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