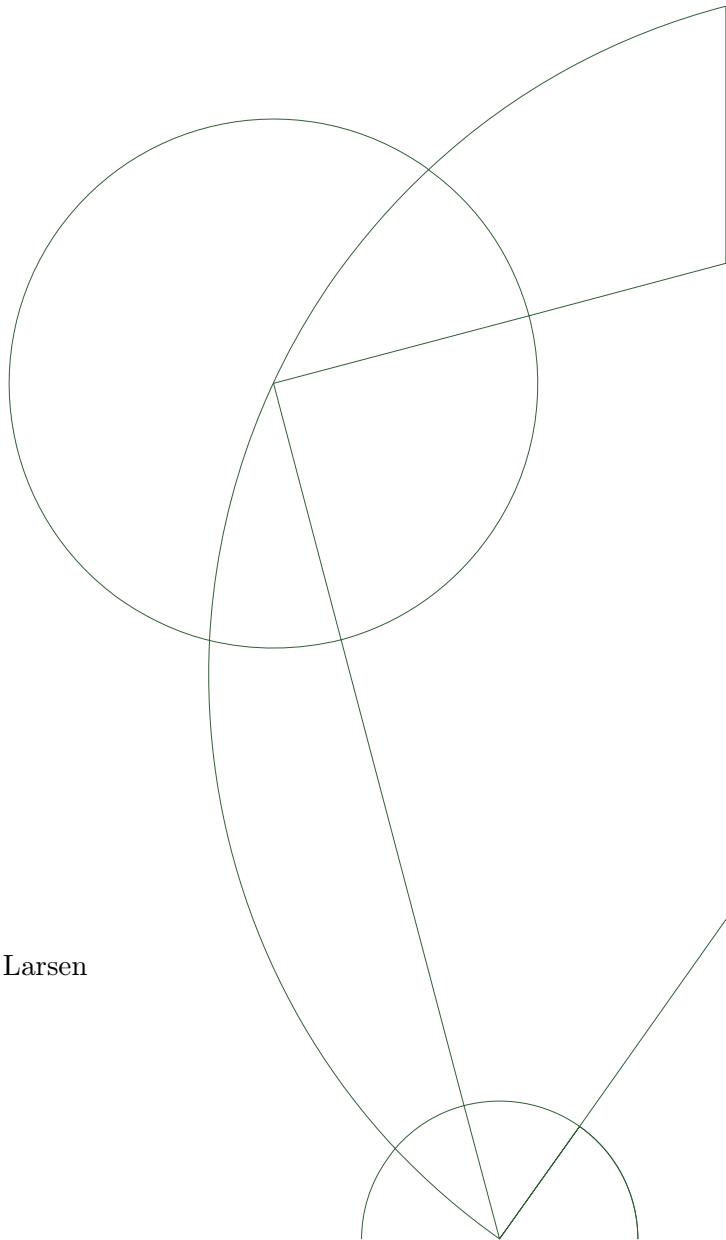
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

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: August 11, 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

August 11, 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 advisors:	Michael Kirkedal Thomsen, Ken Friis Larsen
Submitted:	August 11, 2018

Abstract

The aim of this thesis is to look at cryptography in a reversible computing environment. Reversibility works on the principle that no information can be destroyed during the computation, so any computation can be traced back to the beginning. In a clean setting no information is also unintentionally copied. These features have the implication that any residual information is either known (e.g. null-initialized memory stays zero at the end of the computation, or constants keep their value), or the encryption implementation can be potentially attacked by using the residual information to lower the searched entropy for brute-force decryption.

In an emerging classification of encryption algorithms, we consider *lightweight encryption* algorithms for several reasons: reversible computation has a direct relation to low-power applications, and also the current breed of programming languages is still rather experimental; standard practices have not evolved yet, so preferably simpler algorithms shall be considered first.

Contents

Preface

The present thesis constitutes a 30 ECTS workload and is submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at the University of Copenhagen (UCPH), Department of Computer Science (DIKU).

In addition to this text and its appendices, a GitHub repository is available [2] containing all the related files that are being discussed here.

I would like to thank my girlfriend, my new friends in Copenhagen and mainly my supervisors for their enormous support. I have to emphasize the influence Ken and Michael had over me and my work and their friendly and understanding approach to obstacles on the way. I thank them immensely.

Copenhagen and České Budějovice, Summer 2018

Dominik Táborský

Introduction

1

Cryptography has been with us for decades now, yet only recently we figured out how to reliably code and verify an actual implementation with the *F*/HACL* work [16]. A similar approach takes the *Vale* project [9], 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, general high-level 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 required cryptographic properties.

The range of cryptographic functions is significantly broad, so we are limiting the scope to only lightweight cryptography. The core concepts are the same, *i.e.* side-channel vulnerabilities still apply in the same way, but the algorithms can be a lot simpler. Secondly, lightweight cryptography is interesting in its own sense due to its own application within *Internet of Things (IoT)* systems. These systems are limited in processor performance, power consumption, memory size, connectivity and even reaction times (real-time systems). In such cases, lightweight cryptography is useful in providing enough security even with limited resources.

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 (*e.g.* algorithm-specific magic constants or iterators) 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 and executing it within a virtual environment that tracks secret data (see chapter 5).

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. We also take note of the difficulty to learn and use Vale due to the sheer breadth of details the programmer has to consider, instead of leaving it to the compiler.

1.1 Goals

Our aim is to explore the combination of reversibility and cryptography. Irreversible implementations of crypto algorithms often either suffer from insufficiencies from the security perspective or are difficult to implement correctly. Due to performance being also critical, programmers have to actively fight compilers to make sure their code is safe. Our main goal is to alleviate that in at least some way. Reversible code is limited in how it manages memory - no erasure is allowed, and so deallocations have to be preceded by proper cleanup. Reversible algorithms can follow a much cleaner formal definition that does not leak memory contents outside the scope of those functions.

The second goal is to exploit the bijection of crypto algorithms. Decryption is just an inverse function to encryption, so only one of them needs coding. This lowers the implementation and debugging time demands.

The third goal is getting some practical experience with Janus, finding its strengths and weaknesses and suggesting improvements.

1.2 Problem statement

Classic irreversible implementations of encryption schemes suffer from human errors making them vulnerable to side-channel exploits. Reversible implementation removes one type of side-channel and has potential to alleviate the cost of development and debugging.

1.3 Overview of the chapters

Apart from the Introduction, these are the chapters and their short summaries this thesis builds upon:

Analysis Presents short introduction to cryptography, discusses the practical problems with side-channel vulnerabilities, and explores reversible computing and its implications in the context of cryptography.

Design Talks about how we approach the problem with a selection of algorithms, their expected behaviour and how we can verify a lack of state leakage.

Implementation This chapter focuses on writing the selected algorithms in Janus and the practical experience of it.

Evaluation Here we discuss how we verify our claims, then suggest and perform some experiments to do that. We also look at how we can work around the problem of not having a fully reversible platform.

Reflection and future work We reflect on how well our solution can work in practice, and what else needs to or should be done.

do so

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 those 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 [3] and 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.

Definition

Formally we use the following notation to describe a function enc_S and its inverse dec_S 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 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 often split into two groups[1]:incorrect implementation (software bugs, state-based leakages) and so-called *side-channel vulnerabilities*, *side-channel information leakages* or just *side-channels* for short. The first group consists of attacks on weaknesses in the software implementation, mostly state information leakage via registers and/or memory, *i.e.* not clearing memory containing sensitive data [10]. The second group is classified into several subgroups depending on the type of the attack, *e.g.* power consumption and electromagnetic radiation analyses [18]. A timing-based attack exploits different running times of the algorithm depending on its inputs. Arguably, this kind of attack could be classified as both a software bug, since it can be often circumvented in code, but the source of the problem often lies within the processor itself (instructions themselves being timing-sensitive based on input [12]).

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. Because of that, they tend to be the focus of contemporary research in the area [9] [17].

[1] A word on the classification: the grouping provided here follows other sources. From a second perspective, state-based leakages can also be considered a side-channel, since the mathematical concept of cryptography does not deal with how memory is handled. We can counter such argument by saying that the state, especially the key and the plaintext, should remain secret no matter the code. Nevertheless, the classification is only secondary for our purposes.

Examples

The common issue with state-based information leakage is that either the programmer does not clear the memory that contains the sensitive data (*e.g.* a password) and simply deallocates it, or that they do clear it in the code, but the compiler removes that code because it is considered a dead (ineffective) code.

```
int login()
{
  int ret = 0;
  char *password = ask_for_password();
  //ret = ... use the password
  memset(password, 0, strlen(password));
  return ret;
}
```

In the above example, the memory pointed to by the `password` variable is supposed to be cleared out using the `memset` function, but since the memory is not being used anymore, optimizing compilers remove that call completely [17].

An example of a timing vulnerability is in the case of the RC5 algorithm [12], which uses bitwise rotation, where the number of bits rotated depends on the input data. The vulnerability exists only on some platforms, which differ from the others in the implementation of the rotation instruction. The rotation can be performed in two ways:

1. perform one rotation by n bits,
2. perform n rotations by 1 bit.

This minor change can have drastic effects on security.

Avoidance

There are multiple ways for avoiding side-channel vulnerabilities, but most (if not all) of them require some extra work on the programmer's side. The most time consuming and also error-prone method for avoiding state leakage would be manually clearing memory. Just tracking the used memory would be exhausting. Naturally, the programmer could create a *memory pool*, *i.e.* a block of memory, where the algorithm would store its data. This block would then be erased altogether after the algorithm has finished. Along with that, the programmer would also need to erase the stack and processor registers. A reliable way of doing that is extending a compiler, which is described in [17] and which we also use (see section 3.4 and later 5).

Avoiding a timing side-channel is significantly harder, since it depends on the processor instruction semantics and may depend on the algorithm. There has been some progress with this, and we discuss this in 3.4. Every side-channel is specific, however, and requires specific countermeasures. In case of the power analysis, for example, the actual device has to be designed from the ground up to be resilient.

2.3 Reversibility and its implications

Reversibility is the ability of a program to be executed both forwards and backwards deterministically. Although discussed earlier, first major research

was done by Landauer [14]. Bennett [11] and others continued the research and a first logically reversible language Janus was born ([15], [21]).

Definitions

Consider a reversible program p and some values x and y. Then $[\![p]\!](x) = (y)$ represents[2] the execution of program p with x as a given parameter and y being the result. So in the case of encryption we have:

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

Note that before *enc* represented a function in a mathematical sense, but in this case it is a program composed of some instructions. The $[\![\cdot]\!]$ operator stands for the forward interpretation of the given program. For backward interpretation we use the inversion operator $\mathcal{I}(\cdot)$, which takes a program and inverts it, so that $[\![\mathcal{I}(\cdot)]\!]$ runs the given program backwards. Now we clearly get that:

$$[\![\mathcal{I}(p)]\!]([\![p]\!](x)) = x\,.$$

Cryptography as a reversible embedding

Some cryptographic functions are partial functions, but we can always limit their domains to valid inputs. We will also assume these functions are injective. Altogether, we can assume to work with bijective functions. With this assumption, we can already make one interesting observation:

$$\begin{aligned}[\![enc]\!](k, p) &= (k, c)\\ [\![dec]\!](k, c) &= [\![\mathcal{I}(enc)]\!](k, c) = (k, p)\,.\end{aligned}$$

The implication of that is that we can avoid implementing the decryption functions after having implemented the encryption ones (and vice-versa). In practical experience, this fact is easily observable and immensely useful. It has several advantages:

1. time saved on implementing the inverse function,
2. time saved on debugging it and making sure it actually is the correct inverse
3. easier maintenance and changes
4. smaller source files (and potentially even binary files)

We can see examples in the code developed alongside this thesis; no algorithm required separate encryption/decryption functions.

[2] Here we are using the notation from [13].

Expressive power

Bennett has formalized [11] a reversible Turing machine using deterministic and reversible operations. Quite clearly the expressive power of the reversible Turing machine is not larger than of the irreversible one; irreversibility allows for information erasure, it does not require it. Similarly, the power is not smaller. This can be shown by having the irreversible machine keep a log of its operations and intermediate values, run its algorithm to finish to compute the output, copy the output (without logging) and then run all its operations backwards using the log. The complete proof was also shown by Bennett [11]. Below is a simple example for addition:

```
state: x = 3, y = 5, z = 0, output = 0  // initial state
log: empty
z = (x + y)                             // original algorithm
state: x = 3, y = 5, z = 8, output = 0  // forward run finished
log: set z to (x + y), was 0
output = z                              // copy result to a safe location, log ignored
state: x = 3, y = 5, z = 8, output = 8  // copy saved
log: set z to (x + y), was 0
z = 0                                   // run original algorithm backwards
state: x = 3, y = 5, z = 0, output = 8  // final state with initial input values,
                                        // correct result and no intermediate values
log: empty
```

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[3] 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.

Data transformation

Going back to section 2.3, the second mentioned advantage of the duality of encryption and decryption functions, and similarly with other bijective functions that can be reversed like this, should not be underestimated. The world of programming languages is constantly trying to improve to make the job easier for programmers. Just like some languages are more useful (or at least more common) in certain fields (*e.g.* C for low-level systems, Haskell for parsing, pure computer science and language development, C++ for performance-hungry video games and Java for enterprise applications), reversible languages could create a new such field. Algorithms that perform some kind of data transformation - mathematically bijective or at least injective function application - would be a good fit. The reversibility would then assist with error recovery, for example; it could prove useful in transactional systems, which require the ability to roll back aborted transactions.

Bennett's tricks

In section 2.3 we have already seen the first trick for converting irreversible programs into reversible ones. The trick is fairly simple: to get a result of any function that may create and destroy any number of bits of information, simply provide temporary storage for those bits, run the function without erasing anything until completion, copy out the result and finally run the function backwards, uncomputing the temporary values in the process. This trick was used several times in the cryptographic code developed for this thesis. The *Speck* algorithm is a good example of that, as it requires to perform a fairly complicated key expansion process. There are a few other places where this trick was used, but always only in cases where making a more direct reversible code - that would be equivalent to the originally intended meaning - was either impossible or too difficult.

There also exists a second trick that is not immediately obvious. Having a reversible program p and some input x and corresponding output y, the following holds:

$$[\![p]\!](x) = y[\![\mathcal{I}(p)]\!](y) = x$$

This is just the definition of reversibility. However, the trick is in how this is used: suppose the reversible program p computes its input into output without loss of information. We can store x and y at different memory locations, so that the information within x is duplicated in y. If x is not needed anymore, we can uncompute it by executing the inverse of p with y as its input and x as its output. This method is also described in [11].

[3] By "uncalculating" we mean reversing the program execution, thus going from the output back to the input. See also Bennett's tricks later in this chapter.

state: $a = 3, b = 0, p(x, y) = \lambda x.\lambda y.y+ = (x + 5)$
$[\![p]\!](a, b)$
state: $a = 3, b = 8, \mathcal{I}(p)(x, y) = \lambda x.\lambda y.y- = (x - 5)$
$[\![\mathcal{I}(p)]\!](y, x)$
state: $a = 0, b = 8$

This second trick was used only in the Salsa20 algorithm in the `quarterround` procedure.

Reversible languages

There have been several reversible languages designed (a short summary is provided at the end of [21]), but so far *Janus* has proven most useful. It has an actively developed interpreter in Haskell, another interpreter in JavaScript, which enables running code in a web browser and it was also extended several times [1]. Janus is a (semi-)high-level C-like language, which makes it a fit for comparing implementations with actual C code. Author also has to acknowledge the fact that one of the supervisors is the developer of the interpreter in Haskell, and our cooperation has brought many patches, updates and extensions to it.

Another language that has been considered for this work was RFun, a reversible functional language. However, at the time of writing, RFun was very limited in its abilities, especially when dealing with numbers. Cryptographic code almost always uses some form of bitwise operations and those are not available under Peano number arithmetics; secondly, most of such code is also imperative, not functional. Furthermore, a new development of this language was just under way, so it made sense to stick to a more stable and workable tool.

Design

3

3.1 Avoiding state information leakage

Clearly, reversible code cannot delete information since the output would not be reversible back into its corresponding input. Similarly, no information can be "created", since that would be information deletion when reversed. Information can be duplicated and de-duplicated; one is the inverse of the other, so it is consistent with reversibility. In theory, we can apply this in our bijective crypto algorithms so that for every bit of input information, we output exactly one bit of output information. So assuming the key and plaintext memory locations are taken care of properly with regards to security, then any implementation can avoid leaking information by using only these safe memory locations and uncalculating anything else.

3.2 Example algorithms

Let us select some algorithms to work with. We will start with one of the simplest and still practical encryption algorithms, then move on to some more well known.

Note that these algorithms fit in the **ARX** group as they are based on three operations: **A**ddition, **R**otation and **X**OR operation. This makes them somewhat similar, but they are still different enough to validate our choices; from very simple to complex, block and stream ciphers, old and new. The advantage is that all these three operations are trivially reversible.

TEA

The *Tiny Encryption Algorithm* (TEA) [20] is, per its name, incredibly simple and small. It is so simple that we can even show the complete code for it here; however, since decryption is just a reversal of encryption, we are showing only the encryption procedure:

```
void encrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;           /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
```

```
}
```

Source: Wikipedia

The decryption procedure is analogous. Both procedures take two arguments, a key and some data, and perform the necessary operations; if we avoided using the local variables `v0`, `v1`, `k0` and `k1` and replaced them by what they actually refer to, *i.e.* `v[0]`, `v[1]`, `k[0]` and `k[1]`, the changes would be performed in-place, thus avoiding spilling any secrets anywhere else in memory, except for processor registers.

There are two subsequent extensions, named *XTEA* and *XXTEA*, which improve the cryptographic security. XTEA performs slightly different updates of those `v*` variables, but is the same otherwise. XXTEA finally encrypts a whole block of data instead of just two words. It also uses a few more local variables, but the general approach is identical. We discuss XXTEA in more detail in section 4.1.

RC5

A somewhat more complicated algorithm is *RC5*, which was a candidate for the *AES*. Unlike TEA, RC5 already performs something called *key expansion*. Key expansion often extends the key to a larger data stream; in the case of RC5, the number of encryption rounds is the defining factor in that length. Additionally, some further mathematical operations are applied to it, increasing randomization of key bit distribution. This expansion is performed once for a single key and then re-used, since it only depends on the key and some predefined constants. The encryption itself can then be much simpler, as is the case here. Let us look at RC5 more closely:

```
void RC5_ENCRYPT(uint32_t *v, uint32_t *S, int rounds)
{
   uint32_t i, A = v[0] + S[0], B = v[1] + S[1];

   for(i = 1; i <= r; i++)
   {
      A = ROTL(A ^ B, B) + S[2*i];
      B = ROTL(B ^ A, A) + S[2*i + 1];
   }
   v[0] = A; v[1] = B;
}
```

Sources: Wikipedia, RC5 paper

Just like with TEA, the encryption can be performed in-place. Unlike TEA, the procedure technically asks for a third parameter, a number of rounds; of course that can be removed by using a fixed constant. Also note that the `S` variable represents the expanded key, which brings us to the expansion itself, which is finally interesting (presenting it in a pseudocode for clarity):

```
# w - The length of a word in bits, typically 16, 32 or 64.
# u - The length of a word in bytes.
# b - The length of the key in bytes.
# K[] - The key, considered as an array of bytes (using 0-based indexing).
# c - The length of the key in words (or 1, if b = 0).
# L[] - A temporary working array used during key scheduling. initialized to the key in words.
# r - The number of rounds to use when encrypting data.
```

```
# t = 2(r+1) - the number of round subkeys required.
# S[] - The round subkey words.

# Break K into words
u = w / 8
c = ceiling( max(b, 1) / u )
# L is initially a c-length list of 0-valued w-length words
for i = b-1 down to 0 do:
    L[i/u] = (L[i/u] << 8) + K[i]

# Initialize key-independent pseudorandom S array
# S is initially a t=2(r+1) length list of undefined w-length words
S[0] = P_w
for i = 1 to t-1 do:
    S[i] = S[i-1] + Q_w

# The main key scheduling loop
i = j = 0
A = B = 0
do 3 * max(t, c) times:
    A = S[i] = (S[i] + A + B) <<< 3
    B = L[j] = (L[j] + A + B) <<< (A + B)
    i = (i + 1) % t
    j = (j + 1) % c

# return S
```

Source: Wikipedia

The first loop only makes sure the key is represented in little-endian form. The second loop is just an initialization of the buffer containing the expanded key. The important randomization occurs in the last loop, which seems slightly complicated at first glance. The iteration itself does not use a single variable, but instead uses both `i` and `j`, since both are used as an index for different arrays that may have different lengths. Furthermore, the number of loops is also not set by a single value - it is selected as a maximum of two values. And last but not least, the `A` and `B` variables are being re-set on every loop. No wonder that when we look at the paper establishing RC5, we find this note:

„The key-expansion function has a certain amount of "one-wayness": it is not so easy to determine K from S."

For our purpose, which is re-writing the procedure in a reversible language, that does not sound encouraging. On the other hand, we know that we can make anything reversible using a Bennett's trick at worst. The solution is simple: these statements are actually not mutually-exclusive: the fact is that S is not the only result of the expansion function, because both S and L get updated so that they are dependent on each other. So it is true that getting K from S is not easy, but it is easy to get K from both S and L.

Salsa20 and Chacha20

Salsa20 and Chacha20 are relatively new algorithms. Unlike previous algorithms, they are *stream ciphers*, meaning they do not take the plaintext as an argument, but they generate a stream of random bits, which is then combined (*e.g.* using XOR) with the plaintext. Chacha20 is based on Salsa20 and has improved both performance and security. Code complexity is comparable to

RC5, but it is more synoptic with Chacha20 being even more so than Salsa20 - this becomes clear when reading the Janus code.

Interestingly, both documents for Salsa20 and Chacha20 mention invertibility unlike other documents. Invertibility referred to the mathematical property of bijection rather than reversibility as in our case. Even though invertibility is explicitly mentioned (and required), the code is not reversible directly, as they require some manipulation through a temporary local variable. That is caused by the lack of proper feature of the Janus language rather than some deeper reason. Details, including a suggestion for improvement of Janus, are discussed in section 4.1.

Simon and Speck

Finally we discuss two ciphers designed by NSA, the American intelligence agency. We wanted to see whether reversibility has any potential to reveal crypto algorithm weaknesses. Nothing specific was discovered except a general sense of difficulty implementing them. Although Simon is simpler than its companion, Speck contains multiple uses of Bennett's tricks due to complications with reversing some of the calculations. Even when compared to Salsa20 and Chacha20, which both have more lines of code than Speck, both were easier to write. That experience may be explained with more precise documentation. The fact that both Salsa20 and Chacha20 contain fewer and *„less weird"* [4] occurrences of Bennett's tricks, however, cannot.

3.3 Defining inputs, outputs and expected behaviour

even RC5?

Thanks to assumed bijection, we can define two inputs and two outputs for each of our algorithms.

Also stream ciphers, watch out

3.4 Verification methods

To verify whether our code fits our formal expectations, we have several options to choose from. We will split them into two groups based on user involvement and discuss these separately.

Manual methods

Among the simplest, most demanding and least dependable is experimental behaviour observation and/or testing. We will experimentally show our code cleans up memory after usage.

The second way is showing that cleanup processing using a debugger. That way we can directly point to location in the code where that cleanup occurs. We will use *gdb* to demonstrate memory getting zeroed out.

The third way is manually analysing the actual generated machine code. We will isolate and analyse the instructions related to memory cleanup.

[4] From purely personal experience. I cannot offer formal quantification.

Automatic methods

There are some ways to verify different properties automatically; or to enforce them. Methods like theorem provers, however, tend to be difficult to employ. They often require non-trivial amount of skilled work just to formally prove correctness can be as high or even higher than implementing the algorithm in the first place. We will not be using this method.

citation needed

On a related note, there has been some research made into and real-world application of language-based safe code recently: notably, *F** and its derivative *Low** [8], and the related language *Vale* [4]. Low* has been used in the *HACL* project, which has been integrated into Mozilla Firefox as a crypto library. Vale is a language dedicated to writing verifiable low-level code; its authors also implemented some crypto algorithms, successfully verified their resistance to state-based and timing-based information leakage and even showed some performance gains. Vale has a huge potential, but at the time of writing it also lacks stability, documentation, examples and manuals. Only an unsuccessful attempt to use Vale was made. Details will be discussed later.

discuss later

Another approach is turning the compiler into an ally rather an obstacle that has to be worked-around. Such approach was taken by the authors in a very recent paper [17]. They have implemented an extension to the LLVM/-Clang compiler that adds the ability clearing out registers and stack, and to make a selection between two values in constant time. The C11 standard also added `memset_s`, which the compiler is supposed to guarantee to never remove and so it can be used for clearing out heap memory. We will be using this method in combination with the manual methods to verify results.

Lastly, there are projects that offer tracking of sensitive data (or rather taints) when moved around in memory. These are *taintgrind* and its derivative *secretgrind*, with the latter being specifically designed for crypto algorithms. We will be using *secretgrind* to track tainted data. This will be in combination with the register and stack clearing method, described above, to make sure nothing of the secret computation is preserved.

taints? is this even the right word?

Implementation

4

In this chapter we show our Janus code for the selected algorithms and discuss it. We talk about the conversion from irreversible to reversible and even come up with improvements to Jana, the Janus interpreter.

4.1 Algorithms

TEA

TEA translates to Janus almost directly like in any language, only the local variable allocation and deallocation is Janus-specific.

```
procedure TEA_encipher(u32 data[], u32 key[])
    local u32 delta = 2654435769
    local u32 sum = 0
    iterate int i = 1 by 1 to 32
        sum += delta
        data[0] += ((data[1] * 16) + key[0]) ^ (data[1] + sum) ^ ((data[1] / 32) + key[1])
        data[1] += ((data[0] * 16) + key[2]) ^ (data[0] + sum) ^ ((data[0] / 32) + key[3])
    end
    delocal u32 sum = 32*delta
    delocal u32 delta = 2654435769
```

XTEA is just as easy to translate. With XXTEA, we need to carefully update some local variables. Let us look at the code, starting with C:

```
#define MX ((z>>5^y<<2) + (y>>3^z<<4) ^ (sum^y) + (k[p&3^e]^z))

long xxtea(unsigned long* v, unsigned long n, long* k) {
  unsigned long z = v[n-1], y = v[0], sum = 0, e, delta = 0x9e3779b9;
  long p, q;
  q = 6 + 52/n;
  while (q-- > 0) {
    sum += delta;
    e = (sum >> 2) & 3;
    for (p=0; p<n-1; p++) {
      y = v[p+1];
      z = v[p] += MX;
    }
    y = v[0];
    z = v[n-1] += MX;
  }
  return 0;
}
```

Source: Wikipedia, edited

Note that the code is readable thanks to the MX macro, which is not available with Janus[5].

The problematic parts are the `e`, `y` and `z` variables. They are being reset repeatedly, erasing information. Furthermore, `z` is being set to a value that is dependent on itself and that is not allowed in Janus. Since `e` is set to values dependent only on the variable `sum`, which is altered only at the beginning of the main loop, and then some constants, we know we can recalculate `e` at the end of the loop again. So to update `e` reversibly, we simply subtract the same value it has at the end of the loop.

The problem with `y` and `z` is their dependency on each other[6]. The easy way to overcome this is to use a Bennett's trick. The harder way is figuring out what is actually being updated and with what values. In the case of XXTEA this is still fairly manageable, so let us practice. The first thing we realize is that we can safely replace `y` with whatever data it was set to. We see that it is being set to the value of `v[p+1]`, so we replace all occurrences of y with that. Along with that we also have to partially unroll the inner loop to make sure that the indexed access is within bounds. Then, when we remove `y`, we notice that we don't have to set `z` as nothing but the `MX` macro depends on it, and in that case it simply has the value of the data at the previous index `p`. Again, we replace all occurrences of `z` with `v[p-1]` and pay attention to array bounds.

Finally, the variable `p` also needs care when resetting, but that is straightforward to do according to the inner loop exit.

The resulting Janus code follows the above description:

```
procedure XXTEA_encipher(u32 data[], u32 key[])
    local u32 delta = 2654435769
    local int n = 4 //size(data)
    local u32 sum = 0
    local u32 e = 0
    local int p = 1
    local int q = 6 + 52/n
    from q = 6 + 52/n loop
        sum += delta
        e += (sum / 4) & 3
        data[0] += ((((data[n-1] / 32) ^ (data[1] * 4)) + ((data[1] / 8) ^ (data[n-1] * 16))) ^ ((sum ^ da
        from p = 1 loop
            data[p] += ((((data[p-1] / 32) ^ (data[p+1] * 4)) + ((data[p+1] / 8) ^ (data[p-1] * 16))) ^ ((
            p += 1
        until p = n-1
        data[n-1] += ((((data[n-2] / 32) ^ (data[0] * 4)) + ((data[0] / 8) ^ (data[n-2] * 16))) ^ ((sum ^
        p -= (n-2)
        e -= (sum / 4) & 3
        q -= 1
    until q = 0
    delocal int q = 0
    delocal int p = 1
    delocal u32 e = 0
    delocal u32 sum = (6+52 / (u32)n)*delta
    delocal int n = 4 //size(data)
    delocal u32 delta = 2654435769
```

[5]We could use some preprocessor like *m4* or *cpp* like C does, but the Janus interpreter would not translate that into C without recognizing (or ignoring) it itself. It is, therefore, a good suggestion for Jana improvement.

[6]This dependency is eventual with `y` and immediate with `z`.

RC5

Translating RC5 into Janus was difficult and required extra work in designing the reversible code. The encryption function is simple, but the key expansion function has grown considerably during translation even when partially altered - the last loop has a simplified exit condition.

polish RC5. L is not initialized because of the missing first loop. encryption function has wrong api, it should do changes in-place

Salsa20 and Chacha20

In case of Salsa20, we are not taking existing C code or pseudocode and translating it into Janus. We are following a precise mathematical formulation that is presented in the specification [7]. As a result, we do not compare our implementation with C code anymore. We focus solely on Janus instead. We also focus on the difficult or annoying parts of the implementation by suggesting enhancements to Janus.

First enhancement The first issue appeared with the `quarterround` function. Initially, a misunderstanding of the specification resulted in applying the second Bennett's trick like this:

```
procedure quarterround(u32 seq[4])
   local u32 tmp_seq[4]
   call quarterround_bt(seq, tmp_seq)
   seq[0] <=> tmp_seq[0]
   seq[1] <=> tmp_seq[1]
   seq[2] <=> tmp_seq[2]
   seq[3] <=> tmp_seq[3]
   uncall quarterround_bt(seq, tmp_seq) // 2nd Bennett's trick
   delocal u32 tmp_seq[4]
```

Noticing the error, the `quarterround` procedure was reimplemented to make its computations in-place. Now only one temporary variable is required, and that is only due to the feature lacking within the Janus language. The suggestion is to implement functions, *i.e.* procedures that return a value. Let us consider this extension with an example:

```
// classic Janus
procedure multiply_proc(int a, int b, int c)
  c += a * b

// Proposed extension
function multiply_func(int a, int b)
  return a * b

// Usage of the procedure, computes result += a * b * c
procedure multiply_three(int a, int b, int c, int result)
  local int tmp = 0
  call multiply_proc(a, b, tmp)
  call multiply_proc(tmp, c, result)
  uncall multiply_proc(a, b, tmp)
```

```
  delocal int tmp = 0


// Usage of the function
procedure multiply_three(int a, int b, int c, int result)
  result += call multiply_func(call multiply_func(a, b), c)
```

Obviously the procedure `multiply_three` would also be a function, but we disregard that for this example.

These functions would be just a syntactic sugar compared to what the programmers does with procedures. The extension then would follow these steps:

1. automatically allocate a hidden temporary variable, local to the caller, for storing the returned value
2. add that variable as another parameter
3. replace the `return` statement with adding the value to the new parameter (using the `+=` operator)
4. add corresponding uncall for clearing that hidden variable in the caller

Note that with these steps followed, the syntactic sugar would in fact unroll into this:

```
// Usage of the procedure, computes result += a * b * c
procedure multiply_three(int a, int b, int c, int result)
  local int hidden_tmp1 = 0
  local int hidden_tmp2 = 0
  call multiply_proc(a, b, hidden_tmp1)
  call multiply_proc(hidden_tmp1, c, hidden_tmp2)
  result += hidden_tmp2
  uncall multiply_proc(hidden_tmp1, c, hidden_tmp2)
  uncall multiply_proc(a, b, tmp)
  delocal int hidden_tmp2 = 0
  delocal int hidden_tmp1 = 0
```

This code is not as efficient as the hand-written equivalent above, however, in the case of the `quarterround` function in Salsa20, there would be no performance loss. Consider this piece of code, with both the hand-written form and the one with the suggested extension:

```
  // z1 = y1 ^ ((y0 + y3) <<< 7)
  tmp += seq[0] + seq[3]
  call rotate_left_u32(tmp, 7)
  seq[1] ^= tmp
  uncall rotate_left_u32(tmp, 7)
  tmp -= (seq[0] + seq[3])

  // z1 = y1 ^ ((y0 + y3) <<< 7)
  seq[1] ^= call rotate_left_u32(seq[0] + seq[3], 7)
```

Following the steps above, the automatically unrolled code would be the same as the hand-written code. This extension would allow for replacing the 22-line procedure with a 5-line function with the same semantics and much better readability.

Second enhancement The obvious second hassle is the way we exchange values between a temporary array and the original array in the cases of the `doubleround` (only Chacha20), and `columnround` and `rowround` (only Salsa20) procedures. This is only caused by the need to access different elements in an array. This is the current form of the code:

```
procedure rowround(u32 seq[16])
   local u32 tmp_row[4]
   ...
   // third row
   tmp_row[0] <=> seq[10]
   tmp_row[1] <=> seq[11]
   tmp_row[2] <=> seq[8]
   tmp_row[3] <=> seq[9]
   call quarterround(tmp_row)
   tmp_row[0] <=> seq[10]
   tmp_row[1] <=> seq[11]
   tmp_row[2] <=> seq[8]
   tmp_row[3] <=> seq[9]
   ...
   delocal u32 tmp_row[4]
```

With this example (and considering the rest of the procedure) we see the need to access 4 elements, in two consecutive ranges. In the next listing we are accessing columns also in two ranges:

```
procedure columnround(u32 seq[16])
   local u32 tmp_col[4]
   ...
   // second row
   tmp_col[0] <=> seq[5]
   tmp_col[1] <=> seq[9]
   tmp_col[2] <=> seq[13]
   tmp_col[3] <=> seq[1]
   ...
   delocal u32 tmp_col[4]
```

We have taken a different approach with Chacha20's `doubleround` procedure. We changed the `quarterround` procedure signature to take the 4 elements as single arguments instead of as a single array. This simplified the `doubleround` procedure significantly:

```
procedure doubleround(u32 seq[16])
   ...
   // third diagonal
   call quarterround(seq[2], seq[7], seq[8], seq[13])
   ...
```

Notice that there is no need for any temporary variable. This approach works, but, to a programmer's eye, it feels too unwieldy. In many other languages, the solution to that is called *array slicing*. That is a technique that allows splitting, cutting or *slicing* parts the array into a new array. This new array is often not a full array with its own copies of the elements, but only a syntactic sugar, just a mapping to the original array. If we wanted to slice the array according to the Chacha20's `doubleround` indexing, we would need either a powerful expression system to describe the complex slices (the diagonals), or a way to pick the elements manually.

```
procedure doubleround(u32 seq[16])
    ...
    // third diagonal
    call quarterround(seq[2,7,8,13])
    ...
```

Chacha20 Chacha20, being only a minor update to Salsa20, does not differ dramatically much from its predecessor. The differences consist of different operation ordering and are in the `doubleround` and the `quarterround` procedures. While Salsa20 has its `columnround` and `rowround` functions, their code is mixed in together in the Chacha20 variant.

chacha20 quarterround differs!

4.2 Simon and Speck

4.3 Lack of leakage and translation

Evaluation

5

In this chapter we attempt to test for the properties discussed in section 1.1. These are the goals again:

1. show lack of state information leakage,
2. exploit the duality of encryption and decryption in a reversible setting,
3. gain practical experience with programming in a reversible language.

We have demonstrated reaching the last goal in previous chapter, where we discussed how we implemented some crypto algorithms, showed some examples and even gave suggestions for improving the Janus language. At the same time, we also reached the second goal, as we implemented only encryption functions. The reader can execute our programs which also include basic keys and data. Uncalling the encryption function is equivalent to calling the decryption function.

The first goal remains and we deal with it here.

5.1 Design

We already mentioned some options for verifying our claim in section 3.4. The best tool for the job would be Vale, but that turned out to be more of a trouble instead of help. There is no doubt Vale can be highly useful, but at the time of writing the maturity of the project is just not at the required level. Even the installation was problematic; we encountered issues that are not usually expected, like folder tree structure incompatibility and case sensitivity in file naming. There were also other, more common difficulties, like dependencies being required at specific version and package management issues. When the installation finally succeeded, the problems did not disappear. The biggest problems are related to the usability: there is very little documentation, there are practically no examples and even the source code seems hardly readable[7]. Furthermore, it seems that to take full advantage of the language, the user also needs to understand the target language (in our case Dafny) and perhaps how the translation is performed. Last but not least, the language is extremely complicated and the effort in learning Vale and implementing an algorithm in it might not be any lower than just using a more classic language and using already established tools and methods for debugging and tuning.

[7] Subjective assessment, the source code for the Vale tool is dense and not commented.

Since we cannot use Vale, the next best option is taint tracking in combination with automatic cleanup of registers and stack memory. Ideally we could use both at the same time, but experiments have shown there is a clash between them. The reason is not perfectly clear, but suspicion lies with how *secretgrind* tracks *libc* calls and the fact that *zerostack* links with *musl-libc*. It should be possible to remove this problem in a production scenario without too much effort. Since *secretgrind* also tracks stack, though, we do not necessarily require this for the evaluation.

For the evaluation we suggest performing an experiment on selected algorithms to approach describing the reality of state leakage. Let us describe the experiment in steps:

Step 1: Translation We start by translating Janus code into C/C++ code. The translation is performed by the Jana interpreter when invoked with the `-c` option. On its own, the C/C++ code can be compiled and executed and performs equivalently to its Janus counterpart. There is no input from outside the program, however.

Step 2: Reference We take a reference implementation of the same algorithm and adjust both the reference and the translated code to read and process data the same way. The reference should form a baseline for our comparison.

Step 3: Taint tracking We compile both programs the same way and input the same data. We compare their execution using *secretgrind* and evaluate their scores in number of bytes potentially leaked.

Step 4: Zerostack substitution *Secretgrind* traces the source of the leaks (taints), and so we can examine them. We then recompile the programs with *zerostack* and re-evaluate the situation.

The last step is somewhat problematic. During the experimentation an issue with the combination of *secretgrind* and *zerostack* appeared. To be fully effective, *zerostack* requires all linked libraries, including `libc`, to be also modified with cleanup code. The author of *zerostack* chose `musl-libc` for its simplicity, which reduces potential problems during compilation. However, it appears `secretgrind` does not cooperate very well with that library, as any attempts to trace binaries, that are linked with *zerostack*-enabled `libc`, results in various errors. At least one of them suggests *secretgrind* is somehow sensitive to the choice of `libc`, even though the original version, *taintgrind*, should not be[8]. This means we will resort to manual examination of the executable binary file.

On a last note, the reason to use *zerostack* at all is just usefulness. At this time, reversible hardware is not available. Reversible platforms, which would simulate reversible hardware, are either not commonly available, or just not practically feasible for any serious application. For our research to be usable in the real world, *i.e.* taking the advantages of reversible embedding out of the reversible-only world, we translate the Janus code and make sure the code keeps the necessary guarantees. *Zerostack* offers us the proper cleanup on every function/procedure exit, thus avoiding information leakage.

[8]Confirmed by its author over e-mail.

5.2 implementations

explain edits to translated code and reference chacha implementation

5.3 Experiments

Step 1: Translation

The code is translated using Jana. There are no manual modifications to make the code compile, however there are some modifications in order to perform the experiment. One set of modifications is done so that we *secretgrind* has tainted memory to track. A second set of modifications is done in compliance with *zerostack*, and those will be explained in the last step. We use the older version of translation due to added complexity in recent Jana versions. Originally Jana used a direct translation to C-style arrays, that was substituted by `std::array` for a short time and then by `std::vector`. Reason for the replacement of C-style arrays was keeping some record of the size of the array. Janus interpreter has a keyword `size`, which returns the size of the array, since it is always known as it does not change. C, on the other hand, uses simple pointers to memory and does not track the size of the allocated block[9]. The reason to stick to the original translation is simplicity in tracking memory using *secretgrind* and the immaturity of the recent changes to Jana.

Step 2: Reference

To have something to check our code against, we download reference implementations from the Internet. We also add some boilerplate code to create executable self-standing programs instead of libraries, and we also add a function for reading from a file. This function uses the `read()` function, since that is simple enough to avoid confusion and complexity when tracked by *secretgrind*. For the same reason we also disable code that prints out the contents of arrays - prior experiments have shown that functions like `printf` and `scanf` manipulate and taint more memory.

Step 3: Taint tracking

XXTEA

XXTEA is the first algorithm for us to examine. While we already talked about TEA and XTEA, they are somewhat simpler and perform their operations in-place, which limits potential leakage. XXTEA, on the other hand, required some work to make the updates in-place. That makes it interesting to compare to classic irreversible implementation.

For the testing, we are using the `TEA.janus` file, containing all three TEA-family algorithms; we only comment out the TEA and XTEA invocations. For the reference implementation, we make minor edits to the version on Wikipedia. For input, we load both key and data for encryption from a file, that will be marked as tainted by *secretgrind*. Note that both the key and the data block take up 16 bytes of memory, so a non-leaking implementation should see exactly

cite or hyperref?

[9] Not related to heap allocations, which do need to track that information internally.

32 bytes in total of tainted memory. Let us look at the *secretgrind* output for the reference code:

```
***(2) (stack)  range [0xffefffdcc - 0xffefffdcf]  (4 bytes)  is tainted
   > (stack) [0xffefffdcc - 0xffefffdcf] (4 bytes): XXTEA-C.c:51:@0xffefffdcc:y
        tainted     at 0x400ABF: xxtea_dec (XXTEA-C.c:62)
                    by 0x400C68: main (XXTEA-C.c:96)

***(1) (stack)  range [0xffefffdd4 - 0xffefffdd7]  (4 bytes)  is tainted
   > (stack) [0xffefffdd4 - 0xffefffdd7] (4 bytes): XXTEA-C.c:51:@0xffefffdd4:z
        tainted     at 0x400A55: xxtea_dec (XXTEA-C.c:61)
                    by 0x400C68: main (XXTEA-C.c:96)

***(1) (stack)  range [0xffefffe10 - 0xffefffe2f]  (32 bytes)  is tainted
   > (stack) [0xffefffe10 - 0xffefffe1f] (16 bytes): XXTEA-C.c:90:@0xffefffe10:key
        tainted     at 0x4F196B0: __read_nocancel (syscall-template.S:81)
                    by 0x400B63: read_data_binary (XXTEA-C.c:77)
                    by 0x400C3E: main (XXTEA-C.c:93)
   > (stack) [0xffefffe20 - 0xffefffe23] (4 bytes): XXTEA-C.c:91:@0xffefffe20:more_dat
        tainted     at 0x400AB7: xxtea_dec (XXTEA-C.c:62)
                    by 0x400C68: main (XXTEA-C.c:96)
   > (stack) [0xffefffe24 - 0xffefffe27] (4 bytes): XXTEA-C.c:91:@0xffefffe24:more_dat
        tainted     at 0x400A29: xxtea_dec (XXTEA-C.c:59)
                    by 0x400C68: main (XXTEA-C.c:96)
   > (stack) [0xffefffe28 - 0xffefffe2b] (4 bytes): XXTEA-C.c:91:@0xffefffe28:more_dat
        tainted     at 0x400A29: xxtea_dec (XXTEA-C.c:59)
                    by 0x400C68: main (XXTEA-C.c:96)
   > (stack) [0xffefffe2c - 0xffefffe2f] (4 bytes): XXTEA-C.c:91:@0xffefffe2c:more_dat
        tainted     at 0x400A29: xxtea_dec (XXTEA-C.c:59)
                    by 0x400C68: main (XXTEA-C.c:96)

Total bytes tainted: 40
```

The printout details three stack ranges. The first two are caused by the `y` and `z` variables, while the last one makes up the actual input for the algorithm, that is the key and the data. The leak is therefore 8 bytes over the expected 32 bytes.

Because our Janus implementation had to avoid these local erasable variables, and thus perform the updates in-place, there are no extra leaks:

```
***(1) (stack)  range [0xffefffe20 - 0xffefffe3f]  (32 bytes)  is tainted
   > (stack) [0xffefffe20 - 0xffefffe2f] (16 bytes): TEA-translated.cpp:150:@0xffefffe
        tainted     at 0x4F196B0: __read_nocancel (syscall-template.S:81)
                    by 0x4015F6: read_data_binary(unsigned int*, unsigned long, unsign
                    by 0x4016F9: main (TEA-translated.cpp:167)
   > (stack) [0xffefffe30 - 0xffefffe33] (4 bytes): TEA-translated.cpp:152:@0xffefffe3
        tainted     at 0x401450: XXTEA_encipher_reverse(unsigned int*, unsigned int*)
                    by 0x40171F: main (TEA-translated.cpp:180)
   > (stack) [0xffefffe34 - 0xffefffe37] (4 bytes): TEA-translated.cpp:152:@0xffefffe3
        tainted     at 0x40137F: XXTEA_encipher_reverse(unsigned int*, unsigned int*)
```

```
                    by 0x40171F: main (TEA-translated.cpp:180)
   > (stack) [0xffefffe38 - 0xffefffe3b] (4 bytes): TEA-translated.cpp:152:@0xffefffe38:more_
        tainted     at 0x40137F: XXTEA_encipher_reverse(unsigned int*, unsigned int*) (TEA-tra
                    by 0x40171F: main (TEA-translated.cpp:180)
   > (stack) [0xffefffe3c - 0xffefffe3f] (4 bytes): TEA-translated.cpp:152:@0xffefffe3c:more_
        tainted     at 0x40124E: XXTEA_encipher_reverse(unsigned int*, unsigned int*) (TEA-tra
                    by 0x40171F: main (TEA-translated.cpp:180)

Total bytes tainted: 32
```

It has to be pointed out that the leakage-free result could be achieved the same way originally. The difference is that the opposite is not true: reversible programming does not allow the leaking version. Still this is a fairly simple algorithm, so let us move on.

Chacha20

missing description here, posting just the results for now.

Expected bytes tainted: 64 (16*4)

In the reference version, there is one stack range for all the data. We only show an excerpt from the output with the total number of bytes tainted at the bottom:

```
***(1) (stack)  range [0xffefffd40 - 0xffefffdff]  (192 bytes)  is tainted
Total bytes tainted: 192
```

Our Janus-to-C translated version is more modest: only the expected 64 bytes are marked as tainted. For some reason, *secretgrind* split the data array into 3 sections, but on inspection they do reside right after each other in memory.

```
***(1) (stack)  range [0xffefffdc0 - 0xffefffdff]  (64 bytes)  is tainted
   > (stack) [0xffefffdc0 - 0xffefffdc3] (4 bytes): Chacha20.janus.3.cpp:702:@0xffefffdc0:seq
        tainted     at 0x403743: Chacha20_encrypt_forward(unsigned int*, unsigned int*) (Chacl
                    by 0x403C31: main (Chacha20.janus.3.cpp:726)
   > (stack) [0xffefffdc4 - 0xffefffdc7] (4 bytes): Chacha20.janus.3.cpp:702:@0xffefffdc4:seq
        tainted     at 0x403743: Chacha20_encrypt_forward(unsigned int*, unsigned int*) (Chacl
                    by 0x403C03: main (Chacha20.janus.3.cpp:723)
   > (stack) [0xffefffdc8 - 0xffefffdff] (56 bytes): Chacha20.janus.3.cpp:702:@0xffefffdc0:seq
        tainted     at 0x4F196B0: __read_nocancel (syscall-template.S:81)
                    by 0x403A05: read_data_binary(unsigned int*, unsigned long, unsigned int*
                    by 0x403B78: main (Chacha20.janus.3.cpp:711)

Total bytes tainted: 64
```

Step 4: Zerostack substitution

check my zerostack-github issue on this

Zerostack offers three types of operation, differing by what they are based on: *function*, *stack* and finally *call-graph*. We tested all three and found the stack-based operation non-operational due to a crash in the LLVM compiler. The first one, function-based solution, is the simplest one but also reportedly [17]

the most performance-hindering one. The call-graph-based type of operation is the most efficient one, truly erasing only what is necessary. It is also the most complex one. Details of their implementation is left to their original paper [17], but we at least look at the resulting assembly code, which will become useful later.

Manual modifications First we make some modifications so that all our compilation passes execute correctly:

- We comment out the XTEA-* and XXTEA-* functions because we do not need them for this part of the experiment.
- We change the return value of the TEA-* functions from `void` to `int` and append `"return 0;"` at the end of each of them. This is only so that the patching process for the call-graph method works. The patching script also required some fixing: we had to add assembler code that zeroes out the `r8d` register using the `xor` instruction.

```
diff --git a/examples/fpointer/patchme.py b/examples/fpointer/patchme.py
index 680d58f..a06ac95 100755
--- a/examples/fpointer/patchme.py
+++ b/examples/fpointer/patchme.py
@@ -699,7 +699,8 @@ class Dispatcher:
                                                    "EAX"   :       "\x31\xC0",
                                                    "EDX"   :       "\x31\xD2",
                                                    "ESI"   :       "\x31\xF6",


+                                                   "R8D"   :       "\x45\x31\xC0",
                                                    "R9D"   :       "\x45\x31\xC9",
                                                    "R10D"  :       "\x45\x31\xD2",
                                                    # 16bit register
```

- We define a macro for defining sensitive functions, which is used by the call-graph method. We then use that macro to annotate our two TEA-* functions.

```
#define __attr_zerostack __attribute__((annotate("SENSITIVE")))

__attr_zerostack int TEA_encipher_forward(unsigned int *data, unsigned int *key);
__attr_zerostack int TEA_encipher_reverse(unsigned int *data, unsigned int *key);
```

- We comment out the `assert` checks at the end of the TEA-* functions. This is also due to a problem with compilation with *zerostack* enabled.

Compiling and analyzing Starting with a normally compiled[10] binary, we get a normal function epilogue (in this case, it is the `TEA_encipher` function). The output depends on what compiler we use. The output of `g++` looks like this:

[10] In all the shown cases we used the `-O0` option to disable optimizations.

```
400823:       b8 00 00 00 00          mov    $0x0,%eax
400828:       5d                      pop    %rbp
400829:       c3                      retq
```

The first instruction sets the register `eax` to zero, which is then used as the return value. The second one resets the stack frame pointer in register `rbp` and the `retq` instruction performs the return from the function.

LLVM's `clang` outputs this:

```
400840:       31 c0                   xor    %eax,%eax
400842:       5d                      pop    %rbp
400843:       c3                      retq
400844:       66 66 66 2e 0f 1f 84    data16 data16 nopw %cs:0x0(%rax,%rax,1)
40084b:       00 00 00 00 00
```

Similarly to what g++ produces, we start by clearing `eax`, continue with resetting stack frame pointer and exit with a `retq`. The last line contains a multi-byte `nop` instruction and only serves as a padding. That is also produced by gcc, but only with some optimization enabled.

Now we utilize the function-based cleanup:

```
400860:       31 c0                   xor    %eax,%eax
400862:       5d                      pop    %rbp
400863:       50                      push   %rax
400864:       48 31 c0                xor    %rax,%rax
400867:       b9 45 00 00 00          mov    $0x45,%ecx
40086c:       48 8d bc 24 d8 fd ff    lea    -0x228(%rsp),%rdi
400873:       ff
400874:       f3 48 ab                rep stos %rax,%es:(%rdi)
400877:       58                      pop    %rax
400878:       48 c7 44 24 f8 fc ff    movq   $0xfffffffffffffffc,-0x8(%rsp)
40087f:       ff ff
400881:       48 31 c9                xor    %rcx,%rcx
400884:       31 d2                   xor    %edx,%edx
400886:       89 c0                   mov    %eax,%eax
400888:       c3                      retq
400889:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

The original instructions are still there, but just before the function return we see 10 instructions which perform the cleanup. There are 3 `xor` instructions that take a single register twice, thereby clearing it. The rest is slightly chaotic. `rax` gets pushed to stack for later, then cleared. The three following instructions, `mov`, `lea` and `rep stos` clear the stack by setting a counter, setting the starting address and repeatedly writing zeroes, respectively[11]. Then we reload `rax` from stack and overwrite that position on stack with -4[12]. Finally, the strange-looking `mov %eax,%eax` instruction clears the higher 32 bits of the `rax`

[11] 0x45 * 8 = 0x228, *i.e.* it writes 0x45-times 8 bytes to overwrite the whole stack @[%rsp-0x228, %rsp].

[12] Reason for that value is unknown. Reading the source revealed the intent of overwriting the pushed register, but no explanation for that value.

register. Clearly this function epilogue correctly clears all registers and used stack so it is usable for our purposes.

Now the last option we consider is the call-graph-based approach, which is performed in two phases, where the first one is the compilation itself. The second one is a patching process, which we already mentioned in relation to this method. The first phase generates most of the final binary code, but the epilogue contains a placeholder for the real cleanup. It also generates a file with information based on the call graph built from the source. The second phase then reads that information and replaces the placeholder with proper epilogue cleanup. There are technical reasons for this division, mainly based on the shortcomings of `clang` compiler, but those are not important here.

Below is the unpatched epilogue with the placeholder cleanup:

```
4006b0:       31 c0                   xor    %eax,%eax
4006b2:       5d                      pop    %rbp
4006b3:       50                      push   %rax
4006b4:       48 31 c0                xor    %rax,%rax
4006b7:       b9 67 45 23 01          mov    $0x1234567,%ecx
4006bc:       48 8d bc 24 f0 cd ab    lea    -0x76543210(%rsp),%rdi
4006c3:       89
4006c4:       f3 48 ab                rep stos %rax,%es:(%rdi)
4006c7:       58                      pop    %rax
4006c8:       48 c7 44 24 f8 fc ff    movq   $0xfffffffffffffffc,-0x8(%rsp)
4006cf:       ff ff
4006d1:       66 66 90                data16 xchg %ax,%ax
4006d4:       66 66 90                data16 xchg %ax,%ax
4006d7:       66 66 90                data16 xchg %ax,%ax
4006da:       66 66 90                data16 xchg %ax,%ax
4006dd:       66 66 90                data16 xchg %ax,%ax
4006e0:       66 66 90                data16 xchg %ax,%ax
4006e3:       66 66 90                data16 xchg %ax,%ax
4006e6:       66 66 90                data16 xchg %ax,%ax
4006e9:       66 66 90                data16 xchg %ax,%ax
4006ec:       66 66 90                data16 xchg %ax,%ax
4006ef:       66 66 90                data16 xchg %ax,%ax
4006f2:       66 66 90                data16 xchg %ax,%ax
4006f5:       66 66 90                data16 xchg %ax,%ax
4006f8:       66 66 90                data16 xchg %ax,%ax
4006fb:       66 66 90                data16 xchg %ax,%ax
4006fe:       66 66 90                data16 xchg %ax,%ax
400701:       66 66 90                data16 xchg %ax,%ax
400704:       66 66 90                data16 xchg %ax,%ax
400707:       66 66 90                data16 xchg %ax,%ax
40070a:       66 66 90                data16 xchg %ax,%ax
40070d:       c3                      retq
40070e:       66 90                   xchg   %ax,%ax
```

There are similarities to the previous method. We see the code starts the same way, only the counter and starting address for the stack zeroing look suspiciously human-produced. Our suspicion is correct, they will be replaced.

Below stack cleanup we see exactly 20 repetitions of a nop-equivalent instruction[13]. These will also be replaced, and we will see how right below:

```
4006b0:       31 c0                   xor    %eax,%eax
4006b2:       5d                      pop    %rbp
4006b3:       50                      push   %rax
4006b4:       48 31 c0                xor    %rax,%rax
4006b7:       b9 16 02 00 00          mov    $0x216,%ecx
4006bc:       48 8d bc 24 50 ef ff    lea    -0x10b0(%rsp),%rdi
4006c3:       ff
4006c4:       f3 48 ab                rep stos %rax,%es:(%rdi)
4006c7:       58                      pop    %rax
4006c8:       48 c7 44 24 f8 fc ff    movq   $0xfffffffffffffffc,-0x8(%rsp)
4006cf:       ff ff
4006d1:       31 d2                   xor    %edx,%edx
4006d3:       90                      nop
4006d4:       48 31 c9                xor    %rcx,%rcx
4006d7:       89 c0                   mov    %eax,%eax
4006d9:       90                      nop
4006da:       66 66 90                data16 xchg %ax,%ax
4006dd:       66 66 90                data16 xchg %ax,%ax
4006e0:       66 66 90                data16 xchg %ax,%ax
4006e3:       66 66 90                data16 xchg %ax,%ax
4006e6:       66 66 90                data16 xchg %ax,%ax
4006e9:       66 66 90                data16 xchg %ax,%ax
4006ec:       66 66 90                data16 xchg %ax,%ax
4006ef:       66 66 90                data16 xchg %ax,%ax
4006f2:       66 66 90                data16 xchg %ax,%ax
4006f5:       66 66 90                data16 xchg %ax,%ax
4006f8:       66 66 90                data16 xchg %ax,%ax
4006fb:       66 66 90                data16 xchg %ax,%ax
4006fe:       66 66 90                data16 xchg %ax,%ax
400701:       66 66 90                data16 xchg %ax,%ax
400704:       66 66 90                data16 xchg %ax,%ax
400707:       66 66 90                data16 xchg %ax,%ax
40070a:       66 66 90                data16 xchg %ax,%ax
40070d:       c3                      retq
40070e:       66 90                   xchg   %ax,%ax
```

First thing we notice is the amount of overwritten stack is much higher at 4272 bytes. That value is pretty large and makes this method more inefficient than the function-based one we discussed before, in contrast to claims in the paper [17]. This value is overestimated and composes of three components: a signal handler (4096 bytes) offset, a red zone offset (128 bytes), and some pre-calculated stack usage (48 bytes). Perhaps a there could be some savings in future improvements of *zerostack*.

We also take note that a subset of the 20 `xchg` instructions were replaced. The substituting code is exactly the same code as in the function-based method.

[13]It is supposed to exchange the value between the `ax` register and itself, therefore it has no effect.

Demonstration of execution To complete the experiment, we list a trace showing the cleanup. The reference code for XXTEA will serve as the information leaking example and `gdb` as the examination tool. We execute the program and stop at the last line, `return 0;`. We set the debugger to repeatedly display the contents of the memory where the `y` and `z` local variables reside[14].

```
(gdb) p &y
$3 = (unsigned int *) 0x7fffffffde40
(gdb) p &z
$4 = (unsigned int *) 0x7fffffffde44
(gdb) display *((unsigned int *) 0x7fffffffde40)
3: *((unsigned int *) 0x7fffffffde40) = 792968749
(gdb) display *((unsigned int *) 0x7fffffffde44)
4: *((unsigned int *) 0x7fffffffde44) = 2860056034
```

We keep stepping instructions until the `rep stos` instruction and observe the output:

```
B+ |0x400901 <xxtea_enc+369>        pop    %rbp
   |0x400902 <xxtea_enc+370>        push   %rax
   |0x400903 <xxtea_enc+371>        xor    %rax,%rax
   |0x400906 <xxtea_enc+374>        mov    $0x49,%ecx
   |0x40090b <xxtea_enc+379>        lea    -0x248(%rsp),%rdi
  >|0x400913 <xxtea_enc+387>        rep stos %rax,%es:(%rdi)
   |0x400916 <xxtea_enc+390>        pop    %rax
   |0x400917 <xxtea_enc+391>        movq   $0xfffffffffffffffc,-0x8(%rsp)

4: *((unsigned int *) 0x7fffffffde44) = 792968749
3: *((unsigned int *) 0x7fffffffde40) = 2860056034
```

At some point the output changes the value of the variables to 0. We keep stepping until the `ret` instruction to confirm:

```
   |0x400917 <xxtea_enc+391>        movq   $0xfffffffffffffffc,-0x8(%rsp)
   |0x400920 <xxtea_enc+400>        xor    %rdx,%rdx
   |0x400923 <xxtea_enc+403>        xor    %rsi,%rsi
   |0x400926 <xxtea_enc+406>        xor    %rcx,%rcx
  >|0x400929 <xxtea_enc+409>        retq

4: *((unsigned int *) 0x7fffffffde44) = 0
3: *((unsigned int *) 0x7fffffffde40) = 0
```

Another alternative A recent standard extension, *C11*, has added a new function `memset_s` which, unlike its original variant `memset`, is guaranteed **not** to be removed by an optimizing compiler. In theory, we could use it as a substitute for *zerostack*, except for two shortcomings:

1. it does not and cannot erase registers, and

[14] Displaying `y` and `z` by variable name does not work as they go out of scope at the end of the function.

2. the programmer would have to make sure it overwrites the right memory range, **manually**.

The first point may not be too much of an issue since there is a general shortage of general-purpose registers on x86 machines and thus they get re-used quickly. However, erasing stack can turn difficult: the programmer would have to know the location of the stack and its correct size. If the sensitive function is being updated during development and/or maintenance, the programmer would need to make sure the erasure is still correct. On the other hand, the `memset_s` function can be easily used to zero out data on the heap, and secondly it is supported by a standard and is much more likely to be supported by any compiler.

Conclusion of the experiment We have shown *zerostack* is effective and that it erases local data as declared and predicted. Therefore we suggest using it or some analogous tool for erasing sensitive data from stack and registers. We also suggest using `memset_s` in complement. We can recommend using *zerostack* as a substitution for a proper reversible platform if the goal is avoiding information leakage.

6 Reflection and future work

add what's required for usability: random number generators, nonce generators

Related Work

7

obvious

Conclusion

8

Discussion of results.

8.1 Contributions

These are the contributions from work on this thesis:

- We have shown reversible computing has other interesting and valuable uses other than power consumption savings and quantum computing applications; these are better code security and lower development costs for bidirectional algorithms.
- We have suggested new enhancements and improvements to Janus or its interpreter Jana. Some of these were implemented alongside the research. We have also found and helped fixing several bugs in Jana. 16 is currently the number of reported issues with Jana, more issues were discussed in person.
- We have published our findings in our paper [19], which has been accepted to the Reversible Computation Workshop 2018. We have also presented our research at the *Öresund Security Day Spring 2018* conference.
- All the work is freely accessible on Github [2]. This includes the algorithms, supporting files like other source codes, binaries and listings, this thesis source files and the wiki resources.
- what else?

Bibliography

[1] Janus website at diku.

[2] Lightweight crypto in reverse. Github repository.

[3] Speed comparison between symmetric and asymmetric encryption.

[4] *Vale: Verifying High-Performance Cryptographic Assembly Code* (August 2017), USENIX.

[5] 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.

[6] Bernstein, D. J. Chacha, a variant of salsa20.

[7] Bernstein, D. J. Salsa20 specication.

[8] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Hritcu, C., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., Wang, P., Béguelin, S. Z., and Zinzindohoué, J. K. Verified low-level programming embedded in F. *CoRR abs/1703.00053* (2017).

[9] Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K. R. M., Lorch, J. R., Parno, B., Rane, A., Setty, S., and Thompson, L. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 917–934.

[10] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 22–22.

[11] H. Bennett, C. Logical reversibility of computation. 525 – 532.

[12] 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.

[13] Jones, N., K. Gomard, C., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* 01 1993.

[14] Landauer, R. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development 5*, 3 (July 1961), 183–191.

[15] Lutz, C. Janus: a time-reversible language. Letter to R. Landauer.

[16] Protzenko, J., Zinzindohoué, J.-K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., and Swamy, N. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages 1*, ICFP (2017), 17:1–17:29.

[17] Simon, L., Chisnall, D., and Anderson, R. What you get is what you C: Controlling side effects in mainstream C compilers. In *3rd IEEE European Symposium on Security and Privacy (EuroS&P)* (2018).

[18] Spreitzer, R., Moonsamy, V., Korak, T., and Mangard, S. Sok: Systematic classification of side-channel attacks on mobile devices. *CoRR abs/1611.03748* (2016).

[19] Táborský, D., Larsen, K. F., and Thomsen, M. K. Encryption and reversible computations (unpublished yet). A work-in-progress paper, admitted for the Reversible Computation Workshop 2018. Will be published in the LNCS volume.

[20] Wheeler, D., and Needham, R. Tea: a tiny encryption algorithm.

[21] Yokoyama, T. Reversible computation and reversible programming languages. *Electronic Notes in Theoretical Computer Science 253*, 6 (2010), 71 – 81. Proceedings of the Workshop on Reversible Computation (RC 2009).

[22] Yokoyama, T., and Glück, R. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 2007), PEPM '07, ACM, pp. 144–153.

The extra

A