License codes algorithms

Giuliano Bertoletti
Homepage: `http://www.webalice.it/giuliano.bertoletti`
E-mail: *gbe32241@libero.it*

January 16, 2010

Contents

Preface

This document explains how the algorithms used by the applications *DRegZ* and *QRegZ* work. These tools are used for generating license codes. It's not however a user's manual that describes how to use the applications, the main topic here is mathematics. A solid grasp of abstract algebra and finite fields in particular may help the reader to follow the presentation. Scattered along the document are references to books and papers useful for a deeper coverage of the topics.

The latest version of this document and source code is available at:
`http://www.webalice.it/giuliano.bertoletti/lca.html`

Chapter 1

License codes

1.1 Introduction

License codes are short strings of characters (typically from 4 to 30 or more) which are used to control access to a multitude of systems. In this document we'll consider mainly software applications as such systems, but the concept is broader and can be applied to many other enviroments: i.e. currency notes, train tickets, electronic door locks and so on.

When considering software applications running on a computer, the main idea is that the developers and the company which create the software tend to produce a single prototype for practical matters and packaging costs; each sample might be personalized afterwards.

Consider for example a large software house which invests a lot of money in creating an entertainment multimedia DVD-ROM title such an encyclopedia or a computer game; after an extensive testing of the prototype, the master disk is delivered to a factory where thousand of items are pressed. Clearly the content of all DVDs is the same for they're all derived from the same master; customization is simply not possible or very limited at best at this stage. Once the product is ready to be delivered to the shelves however, it's desirable for the company to label each copy and keep track of them, as far as possible. This is where license codes come into play.

Although in general the company cannot bind each buyer to a particular copy of the product, at least not without recurring to extreme measures like forcing the customer to reveal his/her identity, it can keep generic statistics which may help them to control upgrades and take some sort of remedies in case illegal copies leak out on the net.

A license code might then be printed on the backcover of the DVD box inside the package or e-mailed directly to the customer if the product is sold electronically. The user is typically required to type this code in order to install and/or activate the program.

It is important for the code to be reasonably short in order to copy it directly

from the backcover or speak it over a phone line without too much effort. This means that acceptable codes are 20-30 characters long at most.

Longer codes (hundreds of characters or more) may only be used if the product is sold via Web, in this case the license is e-mailed directly to the user which may copy and paste the text in the target application dialog box. However this consistutes a limitation for sales, because the user is required to have internet access and interact directly with the vendor.

License codes allow also companies to better control their program updates. For example immagine someone buys your application from a website. Some days after the purchase the license code leaks over the net. If it becomes public, the vendor cannot do much to prevent users from installing and using any versions of the program which were released before the leakage. He can however blacklist that license in future updates, so that any users who fraudolently used that code will be unable to upgrade the application further.

In some cases the code might also let you identify the buyer but chances to proseucte her are slim. This is a list of possible obstacles:

- She may live in another country where copyright laws are not so strict or properly enforced.
- There are cases of fraud where someone buys something with an account of someone else who does know nothing about the transaction.
- The real customer pays a homeless person to buy the program on her behalf.
- The customer who brought the application may claim someone stole it.
- If the program is purchased directly from the shelf in a store, it's hard to know who brought it.

So in the end it's better to prevent or minimize such a loss before, rather than reacting afterwards.

1.2 Cracks and Keygens

Software applications are sequence of numbers. Even if they may appear to a user as a complex interaction of graphics and sounds, from the machine stand point they are no more, no less than a sequence of bytes (although a very long one).

An hacker with the proper combination of skill and unmoral attitude can always modify that sequence and force the program to behave differently, no matter how hard you manage to protect it[1]. Under this assumption, it's frequent that skilled people do create simple programs that modify your application in order to circumvent or disable the entitlement checking mechanism. Those

[1] unless you're executing the code in a special environment which protects it from tampering. This would require however expensive hardware and is not for general use

programs are called *cracks*, and people who write them are called *crackers*. Internet is full of those *cracks*; there even exist search engines specific for that topic. An alternative to programs that patch the main executable is the distribution of the modified executables themselves which occurs when encryption of the code is involved and the needed modifications woudn't be localized to a specific area of code. Still the availability of broadband internet makes for an easy transfer of also those type of cracks.

Keygenerators (*keygens* for short) are an even worse threat. They are small programs too, spread with the intent of removing copy protections. They do not modify the main application however, they simply generate license codes as the vendor (or somebody acting on his behalf) would. The malevolent user types in her name (or some fake id, it doesn't matter) and the *keygen* returns a valid license that she can use as if she brought and paid for the application. Later an update can also be easily installed since the software house doesn't know which code she's used to register the application and cannot therefore blacklist it. In other words, if a code emitted by the vendor is leaked into the net, chances are he becomes aware of the misuse and disables it on later versions. But if a code is generated by a *keygen* it remains private and unlocks only that particular installation. Each user registers the application with a different randomly generated code.

Stolen license codes face the same problem as *cracks* in the eye of the (fraudolent) beholder, for once a code becomes public it is easily captured by software producers and blacklisted in future upgrades. So if the user is going to update his application he/she needs also a new code, stolen after that particular software version has been released. Either ways the illegal user has to match the proper software build with the proper *crack*/code (or at least a newer one).

You might then wonder why some people bother to remove copy protections. There are many resons, for example:

- They can.
- They have a lot of spare time.
- They consider the task challenging.
- They need to prove to themselves and to others they're cleverer than the author who designed the protection
- They really don't care how much effort it took to design the program and what damage they do by letting other people stealing it. Since there's no way to prove that a customer who's cheating would have otherwise brought the program, they claim the damage for the company isn't real. Besides the potential damage is done to a company (usually believed to be rich and prosperous) not to real people (forgetting of course that companies are actually made of real people).
- The application always costs too much, no matter what is the price.

- They do it for money

While strong license codes - i.e. license codes for which a *key generator* cannot be easily built - do not solve all the problems, they are a useful way to alleviate the spread out of illegal copies. These algorithms are specifically designed to make the task of creating an illegal *key generator* very hard (hopefully impossible without massive computation resources, which are not generally at the disposal of a single or even a group of *crackers*).

If the cost of creating a single valid, yet illegal, key is higher than the cost of legally buying it and the cost of creating an illegal *key generator* requires thousand of users lending their CPU spare time for the task, then the war of license codes is probably won. Clearly the other problem, the *cracks* floating around, still exists and has to be kept under control.

The aforementioned issues are also present in other non software related systems, although in these circumstances they are less apparent. Consider for example an electronic device which is supposed to grant access to some services. If a valid code is supplied, access is granted, otherwise it's denied. The system clearly stays secure as long as an intruder does not gain direct access to the machine. But suppose he/she eventually manage to steal one of those machines. Is this system still secure? Clearly it depends on how well it's designed. If the validation scheme is poorly designed, the intruder may be able to deduce a pattern and pursue it to create bogus codes which are then treated as valid. There's an extra layer here, the hardware, which adds complexity and make exploitations an harder task, but that alone is not enough to claim the system is theft-proof.

Chapter 2

The General Mathematical Model

We start to analyze our mathematical model by focusing on the decoder. The decoder is a portion of code which validates a given code and outputs a result from a set of possible outcomes. Once we have a clear idea of how the decoder works, we'll try to devise the logic of the encoder, that is the algorithm which takes some bit strings of information regarding the license and computes the license code. Notice that the design that follows is only one of the many which fit our requirements.

2.1 Decoder Scheme

Given a code of m bits which represents our license, we define a function:

$$F\colon \{0,1\}^m \to \mathbb{S} \times \mathbb{T} \tag{2.1}$$

where $\mathbb{S} = \{$*accepted, rejected, blacklisted*$\}$ is the set of three possible outcomes which can be returned by the function and $\mathbb{T} = \{0,1\}^{t_b}$ is t_b feature bits string related to a particular license. Note that $\mathbb{T}$ is meaningful only when the first term evaluates to *accepted*.

We can decompose further the F function into smaller pieces. In particular let:

$$f\colon \{0,1\}^m \to \mathbb{T} \times \mathbb{W} \tag{2.2}$$

where: $n = t_b + w_b$ with $t_b \leq w_b$, $\mathbb{W} = \{0,1\}^{w_b}$ and $n \leq m$ (we'll see later why). w is the set of parity check bit strings. We also define a parity check function:

$$C\colon \mathbb{T}, \mathbb{W} \to \mathbb{S} \tag{2.3}$$

that matches the input payload $\mathbb{T}$ with the parity bits. It turns out that we can express F in terms of f and C. More specifically we have

$$F(\bar{X}) = \langle C_1(\bar{X}), f_1(\bar{X}) \rangle \tag{2.4}$$

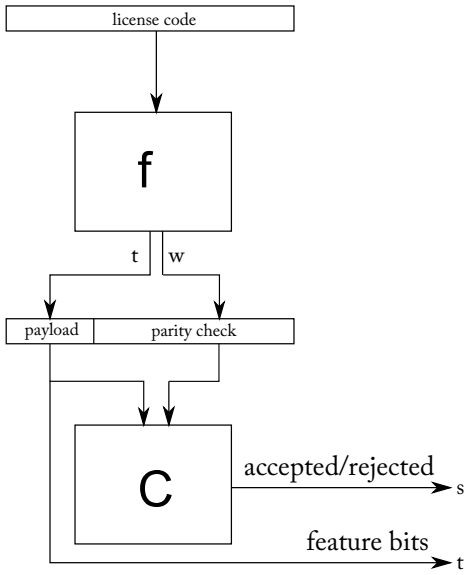

Figure 2.1: general model of a decoder

where we have adopted the convention that $F(\bar{X}) = \langle f_1(\bar{X}), f_2(\bar{X})\rangle$ means that the output of F is just given by the concatenation of the outputs of f_1 and f_2. Figure 2.1 shows how the input license is passed first through the f function, then the payload and parity check strings are evaluated by the C function which estabilishes whether or not the code can be accepted.

2.2 Parity Check Function

The parity check function C is meant to check whether the payload $\bar{t}$ matches the parity check bits $\bar{w}$. What is important in the definition of C is that $t_b \leq w_b$ and that for each $t \in \mathbb{T}$ there exists one and only one $w \in \mathbb{W}$ such that $C(t, w) \in \{accepted, blacklisted\}$[1]. It's also preferable that C is non linear.

One possible way to define C is to represent bit strings $\bar{t}$ and $\bar{w}$ as polynomials defined over $\mathbb{F}_2$ of degree $t_b - 1$ and $w_b - 1$ respectively. We output *accepted* only if:

$$t(X)w(X) = 1 \tag{2.5}$$

over $\Gamma = \mathbb{F}_2(X)/Irr(X) = GF(2^{w_b})$, where Irr(X) is some irreducible polynomial of degree w_b defining the field. In other words, $w(X)$ must be the inverse of $t(X)$ in Γ.

The usual method for computing the inverse of an element in a field is to use the extended euclidean algorithm that we'll now describe briefly. Let $\mathbb{D}$ be

[1] which it won't be possible if it were $t_b > w_b$

an euclidean domain and $a, m \in \mathbb{D}/\{0\}$ two non zero elements of that domain. The extended euclidean algorithm returns, among other things, two constants $s, t \in \mathbb{D}$ such that:

$$as + mt = gcd(a, m) \tag{2.6}$$

If we set m as a defining element of a field $\mathbb{K}$, i.e. either a prime number or an irreducible polynomial in D, it's easy to see that $gcd(a, m) = 1$ for every a, therefore by taking the remainder by m of both sides of (2.6) we get $at = 1$ (since $mt \equiv 0 \bmod m$) or equivalently $t = a^{-1} \in \mathbb{K}$ which is the inverse we're after. Further details on how to compute inverses over finite fields can be found here [1].

Let's now get back to the main topic of this section. The constraint in (2.5) is necessary but may not be sufficient; if a particular code leaks to the public, we might decide to make an exception and output $s = \textit{blacklisted} \in \mathbb{S}$.

Algorithm 1 shows a more general structure of C. Notice that it's slightly differnt from (2.5) logic because it actually compares z against $\bar{w}$ rather than multiplying $t \cdot w$ and checking for the identity.

Algorithm 1 Checks if the parity matches the payload

`INPUT`: payload $\bar{t}$, parity $\bar{w}$
`OUTPUT`: one of the elements in the set $\mathbb{S} = \{accepted, rejected, blacklisted\}$.

```
1: if t ∈ blacklist then
2:     return blacklisted
3: end if
4: z ← CheckParity(t)
5: if w = z then
6:     return accepted
7: else
8:     return rejected
9: end if
```

The size w_b of the input term $\bar{w}$ affects the security of the system. For now let's stick to the observation that a random output of F has probabilty 2^{-w_b} to pass the integrity check.

2.3 Feature Bits

The output $\bar{t}$ of the f function is meant to hold information about the license code. It's up to the developer to estabilish which one; generally however the *license ID* field is always present. The idea is that every license generated has an unique ID and can possibly be used to black list the code, if needed. The first step is therefore to have a rough estimate of how many codes you're going to emit. For example if you plan to emit $2^{20} = 1048576$ different licenses, you need at least 20 bits for the *license ID* field.

Notice that the length of the *license ID* affects the length of the payload (which contains it). On the other hand the size of the payload affects the size n of the output of f, which in turns affects the size m of input[2]: the license code. It's important therefore to find a proper tradeoff between the information stored into the payload and the length of the license code.

For example if an expiration date has to be embedded into the license string, you need to encode it into a certain number of bits and add them to the payload; i.e. you may count the days elapsed from a fixed reference date and have, say, a 16 bit counter keep track of expirations that may be set in the future up to $2^{16} = 65536$ days after that reference date (approximately 179,5 years).

Similary you may need some other bits to encode specific features to be selectively enabled depending on the license purchased. A network application for example could be licensed to a certain number of clients and that number encoded into the feature bits. These bits (see figure 2.1) are returned to the calling routine/application. The encoder and decoder only take care that what is passed to the former is returned by the latter.

Another field which might be included in the payload is the *redundancy*. It will become evident why later in section 2.5 where we discuss the properties of the encoder.

2.4 Core Function f

The core function f shown in figure 2.1 has to be carefully choosen because it's where the reverse engineer will probably focus most of the attention. The main problem is that the value of f should be easily computed by the decoder, but should not be possible - even by inspecting in detail how the decoder works - to figure how out to build an encoder. The solution to this is to use assymmetric cryptography, which is the only way to have the inversion difficult unless something private is known. Of course such information should be kept by the encoder and not embedded into any decoder. One might wonder why not to use RSA to support f then. You embed the public key into the decoder, keep the private key into the encoder and have:

$$f(x) = x^e \, mod \, N \tag{2.7}$$

where e is the public exponent and N is the modulus. Unfortunatelly to be secure, N has to be 1024 bits long or more, which would result into the license string encoding length m to be 1024 bits or more since $0 \leq f(x) < N$ and so does x. Assuming the 5 bits per symbols mentioned earlier, this would result in a license string of no less than 204 characters: unacceptable.

Elliptic curves offer a better tradeoff but still insufficient. We need to operate on arguments at most 128 bits long while maintaining the decoding algorithm relatively simple.

However there exists a family of algorithms that fall under the umbrella of HFE (Hidden Field Equations) which are believed to be less secure than

[2]because we required $n \leq m$

traditional PK schemes, but might be worth using in this context because they offer a decent tradeoff for our needs. We'll analyze them in the next chapter.

2.5 Encoder Scheme

Creating the encoder from the decoder is very simple, provided we do know how to invert f. Suppose we need to emit a license. We build the payload $\bar{t}$ to suit our needs (license ID, redundancy, etc ...) and then compute the parity with:

$$\bar{w} = CheckParity(\bar{t}) \tag{2.8}$$

If we're now able to invert $f(\bar{x}) = \langle \bar{t}, \bar{w} \rangle$ such that $G(\bar{t}, \bar{w}) = f^{-1}(\bar{t}, \bar{w}) = x$ we're done. It's worth noticing, now that we know how the encoder is built, that even if we know how to perform inversion, depending on the f definition, there might not exist an x for every pair $\langle \bar{t}, \bar{w} \rangle$ such that $f(\bar{x}) = \langle \bar{t}, \bar{w} \rangle$. This is where the *redundancy* field of the payload kicks in.

Let ξ be the probability that a solution for a ramdom pair $\langle \bar{t}, \bar{w} \rangle$ does not exist. If we have an r bit *redundany* field, we can generate 2^r different pairs and the probability that none of them can be inverted is given by:

$$P(\xi, r) = \xi^{2^r} \tag{2.9}$$

On the other hand, assuming f is a random mapping between an input set of 2^m elements and an output set of 2^n elements, we have that the probability of not having solutions for a particular $\langle \bar{t}, \bar{w} \rangle$ is given by:

$$Prob\{no\ solutions\} = \left(1 - \frac{1}{2^n}\right)^{2^m} \tag{2.10}$$

which, because of the exponential dependency from n and m, can be approximated by the known result:

$$\lim_{x \to \infty} \left[\left(1 - \frac{1}{x}\right)^x\right]^{2^{m-n}} = (1/e)^{2^{m-n}} \tag{2.11}$$

Then for example, with $n = m$ and $r = 7$ bits of redundancy, we have:

$$P(1/e, 7) = (1/e)^{128} \approx 2.57 \cdot 10^{-26} \tag{2.12}$$

We have thus decreased to a completley negligible figure the probability that the encoder fails to generate a certain license. We obained this by only adding seven bits to the payload!

Another possibility is to implement the redundancy directly in the function f, rather than in the payload, by having $m > n$ so that the $r' = m - n$ bits difference between input and output sizes decrease the probability of having no solutions exactly by the same amount of (2.9).

2.6 Alternatives

The scheme of fig. 2.1 is not unique: there are alternatives. For example the parity check may not be needed if a portion of the output of the f function is matched against an ID supplied externally. This happens when the application takes a fingerprint of the hardware in a challenge response scheme: i.e. the hash is presented to the user, the user sends it to the software house which computes and returns the associated key. Finally the user enters the key, the decoder pass it through f and checks if the output matches the original hash.

Another variant of fig. 2.1 consists in splitting the license L into two parts L_1 and L_2, compute $y = f(L_1)$, $y' = h(L_2)$, where h is a one-way hash function such MD5 or SHA-1 and finally compare y against y'. In this case the encoding scheme would work as follows: a random L_2 is generated with some redundancy embedded, $y' = h(L_2)$ is computed and finally $L_1 = f^{-1}(y')$.

It is clear that despite the scheme adopted, the core function f needs to be a one-way trapdoor function which is invertible only by the one who knows the secret, otherwise the construction of a keygen cannot be prevented.

Chapter 3

The Trapdoor Function

In this chapter we start analyzing various trapdoor functions which can be used to support the scheme shown in figure 2.1.

3.1 *DRegZ*

DRegZ is an application which generates license codes by using an algorithm I derived from Patarin's assymmetric scheme he designed using S-boxes (see [4]).

3.1.1 The Algorithm

The idea is to generate a second degree system of equations defined in $\mathbb{F}_2$ in order to make it easy to compute:

$$\bar{Y} = f(\bar{X}) \tag{3.1}$$

where:

$$\bar{Y} = \langle y_1, y_2, \ldots, y_n \rangle = \langle t_1, \ldots, t_{t_b}, w_1, \ldots, w_{w_b} \rangle \tag{3.2}$$

and

$$\bar{X} = \langle x_1, x_2, \ldots, x_m \rangle \tag{3.3}$$

are respectively an n and m dimensional vector of elements defined over $\mathbb{F}_2$, with $t_b + w_b = n \leq m$. The function f defined in (3.1) is such that can be expressed with a set of n second degree polynomials of the form:

$$\begin{array}{c} y_1 = P_1(x_1, \ldots, x_m) \\ y_2 = P_2(x_1, \ldots, x_m) \\ \ldots \\ y_n = P_n(x_1, \ldots, x_m) \end{array} \tag{3.4}$$

The public key part are the n polynomials, the private key part is an equivalent representation such that it is easy to compute $\bar{X} = f^{-1}(\bar{Y})$.

Let $m_1 < m_2 < \ldots < m_k = m$ and $n_1 < n_2 < \ldots < n_k = n$ a set of $2k$ positive integers. We generate a set of second degree random equations defined over $\mathbb{F}_2$ such that:

$$\begin{aligned}
z_1 &= Q_1(u_1, \ldots, u_{m_1}) \\
z_2 &= Q_2(u_1, \ldots, u_{m_1}) \\
&\ldots \\
z_{n_1} &= Q_{n_1}(u_1, \ldots, u_{m_1}) \\
z_{n_1+1} &= Q_{n_1+1}(u_1, \ldots, u_{m_1}, \ldots, u_{m_2}) \\
z_{n_1+2} &= Q_{n_1+2}(u_1, \ldots, u_{m_1}, \ldots, u_{m_2}) \\
&\ldots \\
z_{n_2} &= Q_{n_2}(u_1, \ldots, u_{m_1}, \ldots, u_{m_2}) \\
&\ldots \\
z_{n_{k-1}+1} &= Q_{n_{k-1}+1}(u_1, \ldots, u_{m_k}) \\
z_{n_{k-1}+2} &= Q_{n_{k-1}+2}(u_1, \ldots, u_{m_k}) \\
z_{n_{k-1}+3} &= Q_{n_{k-1}+3}(u_1, \ldots, u_{m_k}) \\
&\ldots \\
z_{n_k} &= Q_{n_k}(u_1, \ldots, u_{m_k})
\end{aligned} \tag{3.5}$$

In practice it works in blocks, the first n_1 equations are polynomials which operate only on the first m_1 terms and define the first block. The second block is defined by $n_2 - n_1$ polynomials which operate on the first m_2 terms, and so on until the k-th block.
If we choose $m_1, \ldots, m_k$ and $n_1, \ldots, n_k$ properly, we can meet the following criteria:

- **redundancy**: if the number of equations is less than the number of unknowns $n < m$ then the system is underdetermined and we have a greater chance a solution exists.
- **solvability**: if the number of unknowns per set of equations is not too high, we can bruteforce a solution.

Basically given $z_1, \ldots, z_{n_1}$ we can bruteforce a solution for $u_1, \ldots, u_{m_1}$ (first block). Then we substitute the values $u_1, \ldots, u_{m_1}$ in the second block and bruteforce a solution in the remaining unknowns $z_{n_1+1}, \ldots, z_{n_2}$. Proceeding this way, one block a time, we can solve the whole system.

3.1.2 Hiding the System Weak Structure

The idea is to apply two affine transformations $L1$ and $L2$ such that:

$$u_i = L1_i(x_1, \ldots, x_m) \tag{3.6}$$

and

$$y_j = L2_j(z_1, \ldots, z_n) \tag{3.7}$$

Being $L1$ and $L2$ linear and invertible the whole transformation:

$$\bar{Y} = f(\bar{X}) = L2(Q(L1(\bar{X}))) \tag{3.8}$$

remains of second degree, but it's much more difficult to invert.

3.1.3 Encoding a Message

The encoder of *DRegZ* is the most complex part of the program. In order to compute $\bar{X} = f^{-1}(\bar{Y})$ he performs the following steps.

Algorithm 2 Inversion of *DRegZ* core function

`INPUT`: $\bar{Y} = \langle \bar{t}, \bar{w} \rangle$
`OUTPUT`: either $\bar{X}$ or (with small probability) *failure*

1: $\bar{Z} \leftarrow L2^{-1}(\bar{Y})$
2: **if** $Q(\bar{U}) - \bar{Z} = 0$ has solution **then**
3: $\quad \bar{U} \leftarrow Q^{-1}(\bar{Z})$
4: $\quad \bar{X} \leftarrow L1^{-1}(\bar{U})$
5: $\quad$ **return** $\bar{X}$
6: **else**
7: $\quad$ **return** *failure*
8: **end if**

Step two in algorithm 2 is carried out as described before by solving:

$$Q(\bar{U}) - \bar{Z} = 0 \tag{3.9}$$

if possible. In case more than one solution exists, the algorithm simply picks the first found. This doesn't create problems since the decoder has only to apply the direct transformation f when validating, that always works.

3.1.4 Parameters Used

Table 3.1 shows the parameters choosen for *DRegZ* . Brute force is applied 8 times to solve 8 systems of 15 equations in 16 unknowns.

We have therefore $m = 16 \cdot 8 = 128$ and $n = 15 \cdot 8 = 120$ which are the input and output vector size of f respectively.

The input to the decoder is a 25 characters string which is encoded into a 125 bits string; the remaining 3 bits are guessed by computing f at most $2^3 = 8$ times for validating a license. In practice we loop 8 times over the verification routine and append to the supplied 125 bits every combination of the remaining 3 bits. If one of the codes validates, then the license is accepted.

The $CheckParity$ function takes as input a $t_b = 36$ bit payload and generates a $w_b = 84$ bits parity string. Computations are done over the field $GF(2^{84}) = \mathbb{F}_2[X]/Irr(X)$, where $Irr(X) = X^{84} + X^5 + 1$ is the irreducible polynomial generating the field.

Table 3.1: parameters used by *DRegZ*

parameter	value	meaning
k	8	number of blocks
m_i	$16 \cdot i$	unknowns for the i-th block
n_i	$15 \cdot i$	equations for the i-th block
m	128	input size of f
n	120	output size of f
t_b	36	payload size
w_b	84	parity string size
$Irr(X)$	$X^{84} + X^5 + 1$	field generator

The payload is further divided into a 32 bit *license ID* counter and a 4 bits *redundancy* free parameter[1]. Notice also that we applied redundancy both at the payload level and at the f level. The latter is achieved by underdefining the system of equations in (3.5): i.e. $n = 120 < m = 128$.

The *DRegZ* decoder, called SDDecoder, has a simple parity checker that instead of computing the inverse of $\bar{t}$ and comparing it against $\bar{w}$, it multiplies the two polynomials and checks if the result is the identity element of the field.

3.1.5 Deriving Public Key from Private

When a keypair is created, *DRegZ* first generates the private key by building the set of quadratic equations which form Q as defined by equations (3.5). Then the two affine transformation $L1$ and $L2$ are generated and their inverse matrix precomputed, since they are needed to execute algorithm 2. The private key can also be used to evaluate the function f directly by using equation (3.8).

We need now to create the equivalent representation of f shown by equations (3.4) which by construction we know it can be defined by n polynomials of at most second degree. Let's analyze the general structure of such a polynomial, then:

$$y = P(x_1, \ldots, x_m) = \sum_{i,j=1, i<j}^{m} (\alpha_{i,j} x_i x_j) + \sum_{i=1}^{m} \beta_i x_i + \gamma \tag{3.10}$$

where all coefficients and unknowns belong to $\mathbf{F}_2 = \{0, 1\}$. By using the private key we can set $\bar{X} = \langle x_1, \ldots, x_m \rangle$ to the desired value and retrieve the corresponding y using (3.8).

Our goal is to recover the defining coefficients $\alpha_{i,j}, \beta_j$ for $i, j \in [0, n], i < j$ and γ. First we recover γ by setting $\bar{X} = \langle 0, \ldots, 0 \rangle$, so $y = P(0, \ldots, 0) = \gamma$. Then we get:

[1] therefore *DRegZ* does not support any extra feature bits like expiration dates, number of clients and so on

$$\beta_i = P(x_1, \ldots, x_m) + \gamma \tag{3.11}$$

where:

$$x_j = \begin{cases} 1 & \text{if } j = i; \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

In other words we recover each β_i by setting the corresponding i-th element of $\bar{X}$ to 1 and all others to 0. Notice also that $(a + b) = (a - b)$ in $\mathbb{F}_2$, which explains why addition in equation (3.11). The last step is to recover each $\alpha_{i,j}$, but that's an easy ride since we already know γ and all β_i. Indeed we have:

$$\alpha_{i,j} = P(x_1, \ldots, x_m) + \beta_i + \beta_j + \gamma \tag{3.13}$$

where:

$$x_k = \begin{cases} 1 & \text{if } k = i \vee k = j; \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

which again means we set all components of $\bar{X}$ to 0 except those in positions i and j which are set to 1 and use equation (3.8) to compute $f(\bar{X})$. We repeat this operation for all n polynomials and retrieve the public polynomials representation of (3.4). This algorithm is also described in [7].

Finally notice that the operation of deriving the public key from the private is not possible in all PK schemes. For example RSA won't allow you to derive the public exponent from the private one.

3.1.6 Evaluating f with the Public Key

The decoder validates a license using only (3.8). We therefore need an algorithm which performs the computations efficiently in terms of speed and most importantly of space. Our first observation is that in $\mathbb{F}_2, x = x^2$ so equation (3.10) can be rewritten as:

$$y = P(x_1, \ldots, x_m) = \sum_{i,j=1, i \leq j}^{m} (\alpha^*_{i,j} x_i x_j) + \gamma \tag{3.15}$$

where we have suppressed the first degree terms, changed in the remaining summation the constraint $i < j$ to $i \leq j$, set $\alpha^*_{i,i} = \beta_i$ and $\alpha^*_{i,j} = \alpha_{i,j} + \beta_i + \beta_j$, for $i < j$. We can now append another index r to (3.15) to account for the set of n equations:

$$y_r = P_r(x_1, \ldots, x_m) = \sum_{i,j=1, i \leq j}^{m} (\alpha^*_{i,j,r} x_i x_j) + \gamma_r \tag{3.16}$$

with $r \in [1 \ldots n]$.

Algorithm 3 can then be used to compute (3.8); the $\oplus$ operator in $\mathbb{F}_2$ is simply a vector wise exclusive or.

Algorithm 3 decoder evaluates $\bar{Y} = f(\bar{X})$ using public key

INPUT: $\bar{X} = \langle x_1, \ldots, x_m \rangle$
OUTPUT: $\bar{Y} = \langle y_1, \ldots, y_n \rangle$

1: $\bar{Y} \leftarrow \langle \gamma_1, \ldots, \gamma_n \rangle$
2: **for** $i = 2$ to m **do**
3: **for** $j = 1$ to i **do**
4: **if** $x_i x_j \neq 0$ **then**
5: $\bar{Y} \leftarrow \bar{Y} \oplus \langle \alpha_{i,j,1}, \ldots, \alpha_{i,j,n} \rangle$
6: **end if**
7: **end for**
8: **end for**

The only complication is that we do not have n bit registers so in practice we shall break up vector operations with another for loop. Notice also that vectors $\langle \alpha_{i,j,1}, \ldots, \alpha_{i,j,n} \rangle$ can be cleverly packed into sequential memory areas and accessed through a single index since the sequence of i, j is deterministic.

Algorithm 4 encoder packs $\alpha_{i,j,r}$ into $\bar{W}_0, \ldots, \bar{W}_k$

INPUT: $\alpha_{i,j,r}$
OUTPUT: $\bar{W}_0, \ldots, \bar{W}_k$, where $k = \binom{m+1}{2} = \frac{m(m+1)}{2}$

1: $\bar{W}_0 \leftarrow \langle \gamma_1, \ldots, \gamma_n \rangle$
2: $count \leftarrow 1$
3: **for** $i = 2$ to m **do**
4: **for** $j = 1$ to i **do**
5: $\bar{W}_{count} \leftarrow \langle \alpha_{i,j,1}, \ldots, \alpha_{i,j,n} \rangle$
6: $count \leftarrow count + 1$
7: **end for**
8: **end for**

Algorithm 4 shows how the encoder can output single vectors which can then be embedded into the decoder. Once we have the $k + 1$ vectors $\bar{W}_i$ we can easily modify algorithm 3 to optimize the storage (see algorithm 5).

3.1.7 Security Concerns

DRegZ is broken. On January 13th 2010, Andrew Lamoureux cleverly solved a challenge I posted on a reverse engineering site [9] showing that the overlapping sboxes layout is not enough to protect the license scheme against differential cryptanalysis. This is a practical attack that can recover the private information from the public key in a few days, on a single machine processor. Once the private key is recovered the attacker has the same information and therefore is in the same position of the authorized entity emitting licenses.

Algorithm 5 decoder optimized evaluation of $\bar{Y} = f(\bar{X})$ using public key

INPUT: $\bar{X} = \langle x_1, \ldots, x_m \rangle$
OUTPUT: $\bar{Y} = \langle y_1, \ldots, y_n \rangle$

```
1: Ȳ ← W̄_0 {W̄_0, ..., W̄_k is hardwired into the decoder}
2: count ← 1
3: for i = 2 to m do
4:    for j = 1 to i do
5:       if x_i x_j ≠ 0 then
6:          Ȳ ← Ȳ ⊕ W̄_count
7:       end if
8:       count ← count + 1
9:    end for
10: end for
```

The algorithm then offers little security against a knowledgeable user who knows how to attack the scheme mathematically. Luckily the only part that needs a redesign is the equation set (3.5).

3.2 Stepping Up

Now that we laid the basic scheme upon which licenses are built, we examine some variants which will hopefully improve the overall security of the system. As we saw, such security depends on the strenght of the trapdoor function f.

3.2.1 Increasing the Degree of Polynomials

One remarkable consideration about the security of *DRegZ* is that we arbitrarily imposed that polynomials in (3.5) should be at most of second degree, although the security would intuitively improve if we drop that constraint. The drawback is that using equations of higher degree would result into a much larger public key.

The general formula to estimate the number of bits required to store a set of n polynomials of order at most d in m unknowns over $\mathbb{F}_2$ is:

$$N_{bits}(n, m, d) = n \cdot \sum_{i=1}^{d} \binom{m}{i} \tag{3.17}$$

For example for a second degree set of polynomials, substituting in (3.17) $n = 120, m = 128, d = 2$ we get $N_{bits} = 990840$: a little less than 121 KBytes.

DRegZ currently has a public key stored into 33028 32-bit words, which accounts for a total memory use of approximately 129 KBytes. We wasted some space for data structure alignement purposes. If we were to use 3rd degree polynomials, substituting in (3.17) we would get: $N_{bits} = 41955960$,

slightly more than 5Mb, which might be a bit too much to store directly into the program executable. A 4th degree system of equations would cost instead $N_{bits} = 1,322,115,960$ or about 157Mb.

3.2.2 Using Hidden Field Equations

DRegZ structure is essentially built on a scrambled Grőbner basis, in order to keep the overall degree low in (3.5). Our goal here is to devise another mapping of input to output, while keeping the same representation. Instead of hiding the structure into quadratic transformations with reduced unknowns and affine transformations to mix variables, we can use the properties of finite fields[2].

Let $G(x)$ be an irreducible polynomial of degree m defined over $\mathbb{F}_2$. Then $\Lambda = \mathbf{F}_2[x]/G(x)$ is a finite field with 2^m elements. We define:

$$\Upsilon(\bar{Z}) = \bar{\alpha_1}\bar{Z}^{\eta} + \bar{\alpha_2}\bar{Z}^{\eta-1} + \ldots + \bar{\alpha_\eta}\bar{Z} + \bar{\alpha_{\eta+1}} \tag{3.18}$$

where $\bar{\alpha_1}, \ldots, \bar{\alpha_{\eta+1}} \in \Lambda$ are $\eta + 1$ polynomials with degree at most $m - 1$ and $\bar{Z}$ is the unknown vector.

Notice that we sometime use polynomials and vectors interchangeably because they share some properties: i.e. there's a one to one correspondence between coefficients of an m degree polynomial and an $m + 1$ dimensional vector with coefficients/elements defined over the same field, also the + operation has the same meaning of element wise sum. However two vectors can be summed this way only if they have the same size, while this restriction does not exist for polynomials. Polynomials also have a product and thus an exponentiation operatation which vectors do not.

Equation (3.18) if interpreted as a mapping between two vectors of the same dimension fits in the model of equation (3.1) with $n = m$.

So let $\bar{B} \in \Lambda$ be a vector representing the concatenation of the payload and the parity; the equation:

$$\Upsilon(\bar{Z}) = \bar{B} \tag{3.19}$$

may represent a new trapdoor function f where the unknown $\bar{Z}$ represents the license code (see figure 2.1). Then our major concerns regarding to (3.19) are:

1. do a solution always exist ? I.e.: is it true that $\forall \bar{B} \in \Lambda, \exists \bar{Z} \in \Lambda$: $\Upsilon(\bar{Z}) - \bar{B} = 0$?
2. if so, how do we find such solution(s) ?
3. is it possible to find an equivalent representation with a second degree set of equations in the form of (3.4) ?

Let's start with bullet n.3: in any finite field of characterisic p, the transformation: $\bar{Z} \to \bar{Z}^p$ is a linear operation, i.e.:

$$(\bar{a} + \bar{b})^p = \bar{a}^p + \bar{b}^p \tag{3.20}$$

[2] A good introduction to the subject can be found here [3]

This can be proved by observing that the binomial expansion yields coefficients all divisible by p except the first and the last, but any element multiple of p in such a field is zero. More over this works by induction an indefinite number of times, therefore we also have:

$$(\bar{a} + \bar{b})^{p^r} = \bar{a}^{p^r} + \bar{b}^{p^r} \tag{3.21}$$

for any integer $r \geq 0$. In $\mathbb{F}_2$, by definition the characteristic is $p = 2$, therefore (3.20) becomes:

$$(\bar{a} + \bar{b})^{2^r} = \bar{a}^{2^r} + \bar{b}^{2^r} \tag{3.22}$$

which will be useful shortly.

It's important now to recall that being Λ finite dimensional, every linear mapping from Λ to itself can be represented as a matrix. Conversely, matrices yield examples of linear maps. Therefore there exists a square matrix M_r of size m, such that for every $\bar{Z} \in \Lambda$, M_r maps: $\bar{Z} \to \bar{Z}^{2^r}$.

The immediate consequence is that if the equations in (3.18) has coefficients $\alpha_i \neq 0$ associated to only exponents in the form $2^r, r \geq 0$, then $\Upsilon : \Lambda \to \Lambda$ is a linear transformation which can be represented by a matrix. For example:

$$\Upsilon(\bar{Z}) = \bar{\alpha_1}\bar{Z}^{128} + \bar{\alpha_2}\bar{Z}^{32} + \bar{\alpha_3}\bar{Z}^2 + \bar{\alpha_4}\bar{Z} + \bar{\alpha_5} \tag{3.23}$$

is a linear transformation because it's the sum of linear transformations which is also a linear transformation.

On the other hand, if we multiply together the result of two linear transformations, we get a quadratic transformation. So for example:

$$\Upsilon(\bar{Z}) = \bar{\alpha_1}\bar{Z}^{32} \cdot \bar{\alpha_2}\bar{Z}^2 = \bar{\alpha_1}\bar{\alpha_2}\bar{Z}^{34} \tag{3.24}$$

is a quadratic transformation (the coefficient $\bar{\alpha_1}\bar{\alpha_2}$ is constant and operates on $\bar{Z}^{34}$ only linearily) which can be represented by a set of equations of the type (3.16). Consequently (3.18) is at most a quadratic mapping if the non zero coefficients are associated to only exponents in the form $2^r + 2^t$ with $r, t \geq 0$. Another way to think about it is that an the exponent η in its binary representation shall have at most two bits set.

Notice that the procedure to derive the public key from private developed in section 3.1.5 and the evaluation of f in section 3.1.6 still work because we have actually changed the way we created the equations in (3.4) but not their degree or meaning.

Direct evaluation of Υ is also simple if we use a method similar to repeated squaring and reduction used for RSA exponentiation. The only thing to keep in mind here is that we're not dealing with integers but with elements of a finite field. So you feed (3.19) with a polynomial and you get a polynomial in return. This completes bullet n.3.

Regarding bullet n.1, the general answer is no. Unless we work in some extension fields we cannot guarantee that a solution always exists, which is pretty much the same as what happens for infinite fields such as real numbers. For example: $x^2 + 1 = 0$ has no solutions in $\mathbf{R}$ but has two solutions in $\mathbf{C}$, which

is an extension field of $\mathbf{R}$. Another possibility would be to stick to linearity for (3.18) but this is not what we want for security reasons.

We recall however the random model presented in section 2.5 which states that the probability of having no solutions is given by (2.11) and is approximately $1/e$. So by adding some *redundancy* bits to the payload we can lower the chances of having no solutions to a negligible figure as (2.9) states. Notice that respect to the trapdoor used in *DRegZ* here we cannot apply redundancy at the Υ level because the mapping function is an omomorphism (i.e. input and output sets are the same).

Now sit down and relax that comes the most difficult part, bullet n.2, which deserves a subsection of its own.

3.2.3 Root Finding

Let's rewrite (3.18) as:

$$\Psi_{\bar{B}}(\bar{Z}) = \Upsilon(\bar{Z}) - \bar{B} = 0 \tag{3.25}$$

the problem is now to find roots of (3.25), if any.

Suppose now a root exists and let $\bar{\rho_1}$ be such a root, then (3.25) can be rewritten in the form:

$$\Psi_{\bar{B}}(\bar{Z}) = \Omega(\bar{Z})(\bar{Z} - \bar{\rho_1})^{p_1} = 0 \tag{3.26}$$

where p_1 is the multiplicity of $\bar{\rho_1}$ and $\Omega(\bar{Z})$ is a polynomial of degree $\eta - p_1 \geq 0$ such that $\Omega(\bar{\rho_1}) \neq 0$. If we proceed recursively on $\Omega(\bar{Z})$ we observe that it might be futher factored if roots other than $\bar{\rho_1}$ exist. Eventually we come down to the form:

$$\Psi_{\bar{B}}(\bar{Z}) = \Theta(\bar{Z})(\bar{Z} - \bar{\rho_1})^{p_1} \dots (\bar{Z} - \bar{\rho_k})^{p_k} = 0 \tag{3.27}$$

where k is the number of distinct roots and $p_1, \dots, p_k$ their respective multiplicity. Also let $p_1 + \dots + p_k = p \leq \eta$; $\Theta(\bar{Z})$ is either an irreducible polynomial or the product of two or more irreducible polynomials of degree no less than 2. In both cases $\Theta(\bar{Z})$ has no roots in Λ and it's degree is $n - p$.

This is pretty much the same as what happens with real coefficents. For example consider the following polynomial:

$$P(x) = x^5 + 7x^4 - 4x^3 - 68x^2 - 5x - 75 = 0 \tag{3.28}$$

which factors as:

$$P(x) = (x - 3)(x + 5)^2(x^2 + 1) = 0 \tag{3.29}$$

with $x^2 + 1$ being irreducible over $\mathbf{R}$. Similary we have $deg(P) = 5, p_1 = 1, p_2 = 2, \rho_1 = 3, \rho_2 = -5$ and the irreducible polynomial has degree: $5 - 1 - 2 = 2$.

Now it's time to introduce a foundamental result. Let Λ be a finite field with p^m elements[3] and $E = \mathbf{F}_2[X]$ an extension field such that $\Lambda \subset E$. Then in E

$$\bar{Z}^{p^m} - \bar{Z} = \prod_{\bar{\rho} \in \Lambda} (\bar{Z} - \bar{\rho}) \tag{3.30}$$

[3] p must be a prime in order for Λ to be a field

that is: the left side of (3.30) factors as the product of p^m distinct monomials or equivalently that every element of Λ is a root of (3.30).

This is usefull because considering the structure of (3.27) and (3.30) we can set:

$$H(\bar{Z}) = gcd(\bar{Z}^{p^m} - \bar{Z}, \Psi_{\bar{B}}(\bar{Z})) = (\bar{Z} - \bar{\rho_1}) \dots (\bar{Z} - \bar{\rho_k}) \tag{3.31}$$

and recover the product of all monomial of degree 1 of (3.27) with multiplicity 1. If you prefer a different point of view, gcd acts as a filter that extracts from $\Psi_{\bar{B}}(\bar{Z})$ all the first degree factors, which are the only ones responsible for its roots. Now let:

$$R(\bar{Z}) = \left(\bar{Z}^{p^m} - \bar{Z}\right) \, mod \, \Psi_{\bar{B}}(\bar{Z}) \tag{3.32}$$

then (3.31) can be further simplyfied to:

$$H(\bar{Z}) = gcd(R(\bar{Z}), \Psi_{\bar{B}}(\bar{Z})) \tag{3.33}$$

because in any euclidean domain:

$$gcd(a, b) = gcd(a - kb, b) \tag{3.34}$$

with a, b, k elements of that domain. But for a proper k we also have:

$$a = kb + r \tag{3.35}$$

with $0 \leq r < b$ or $0 \leq deg(r) < deg(b)$ if we work in a field of dimension $m > 1$ (as for the case above). But r in (3.35) is exactly the definition of remainder of a division used in (3.32), so (3.33) must hold true.

Notice that the algorithm for computing the greatest common divisor of two polynomials with coefficients defined over $GF(2^m)$ is similar to the one used for integers as all the operations used with integers have an equivalent over finite fields with the exception of comparison: $a < b$ is substituted by $deg(\bar{a}) < deg(\bar{b})$ over finite fields of dimension m greater than 1.

Already at this stage, if we're lucky, we could get $deg(H(\bar{Z})) \leq 1$ which would mean (3.25) has either no solutions or one solution in Λ and in both cases we are done.

There's however a third case where $deg(H(\bar{Z})) > 1$ which need to be discussed. First observe that if we're satisfied with a suboptimal algorithm, we may stop here. If (3.25) has more than one root, or even a single root with multiciplity greater than 1, we won't find it, but chances are limited and *redundancy* bits play on our side.

If we however want to do things well, we need to proceed further and break down $H(\bar{Z})$ into smaller factors. In the next subsection we'll do it.

3.2.4 Refining the Factorization

Before continuing with factorization, we need to introduce some well known results about finite fields. Let $F = GF(q)$ be a field with q elements and

$E = GF(q^m)$ an extension field of F with q^m elements. If γ is an element of E then its trace relative to F is defined as follows:

$$Tr_F^E(\gamma) = \sum_{i=0}^{m-1} \gamma^{q^i} \tag{3.36}$$

It can be proved that $Tr_F^E(\gamma) \in F$ is a linear transformation which maps every element of E into an element of the subfield F. More over for every $c \in F$ the equation:

$$Tr_F^E(\gamma) - c = 0 \tag{3.37}$$

has exactly q^{m-1} distinct roots in E.

Back to the root finding refinement, equation (3.30) can be rewritten as:

$$\bar{Z}^{p^m} - \bar{Z} = \prod_{c \in \mathbb{F}_p} \Big[Tr_{\mathbb{F}_p}^{\Lambda}(\bar{Z}) - c \Big] \tag{3.38}$$

because the left side of (3.38) have as roots the p^m distinct elements of Λ while the right side give the same result since for each element $c \in \mathbf{F}_p$ we have p^{m-1} distinct roots. But since $\mathbf{F}_p$ has only p elements, the total number of roots is also $p^{m-1}p = p^m$. Since there can be no other roots and every root appears only once, the two sides must be equal.

On the other hand by (3.31), $H(\bar{Z})$ divides the left side of (3.38), so it must be:

$$\prod_{c \in \mathbb{F}_p} \Big[Tr_{\mathbb{F}_p}^{\Lambda}(\bar{Z}) - c \Big] \, mod \, H(\bar{Z}) = 0 \tag{3.39}$$

which leads to the important relation:

$$H(\bar{Z}) = \prod_{c \in \mathbb{F}_p} gcd \Big(H(\bar{Z}), Tr_{\mathbb{F}_p}^{\Lambda}(\bar{Z}) - c \Big) \tag{3.40}$$

that shows a possible factorization of $H(\bar{Z})$. Equation (3.40) is practical for finite fields of small characteristic as our since the number of terms in the product of sequences is limited.

Example 1. *Let $F = GF(2)$ be the finite field with elements $\{0, 1\}$ and $E = GF(2^4)$ a finite field with 16 elements. Using* (3.38) *we get:*

$$\bar{Z}^{2^4} - \bar{Z} = \big[Tr_{\mathbb{F}}^E(\bar{Z}) - 1 \big] \cdot Tr_{\mathbb{F}}^E(\bar{Z}) \tag{3.41}$$

Using (3.36) *and observing that $2\bar{Z} = \bar{Z} + \bar{Z} = \bar{Z} - \bar{Z} = 0$ in every field of characteristic 2, we get from the right side of* (3.41)*:*

$$\Big[\bar{Z} + \bar{Z}^2 + \bar{Z}^{2^2} + \bar{Z}^{2^3} - 1 \Big] \cdot \Big[\bar{Z} + \bar{Z}^2 + \bar{Z}^{2^2} + \bar{Z}^{2^3} \Big] = \bar{Z}^{16} - \bar{Z} \tag{3.42}$$

with all coefficients cancelling out except two, in agreement with (3.38)*.*

Unfortunatelly (3.38) does not cover all cases and we need to further investigate the issue. It's possible indeed that one or more of the terms in (3.38) are trivial factors of $H(\bar{Z})$, i.e. elements of Λ, that happens when for some $c \in \mathbb{F}_p, Tr^{\Lambda}_{\mathbb{F}_p}(\bar{Z}) \equiv c \, mod \, H(\bar{Z})$.

To break this enpasse we need an auxiliary polynomial. Let $\bar{V}$ be an element of $\Lambda = GF(p^m)$ so that $\langle 1, \bar{V}, \bar{V}^2, \ldots, \bar{V}^{m-1} \rangle$ is a basis of Λ over $\mathbb{F}_p$. Then if we replace $\bar{Z}$ in (3.38) with $\bar{V}^j \bar{Z}$, for $j \in [0, \ldots, m-1]$ we get:

$$(\bar{V}^j)^{p^m} \bar{Z}^{p^m} - \bar{V}^j \bar{Z} = \prod_{c \in \mathbb{F}_p} \Big[Tr^{\Lambda}_{\mathbb{F}_p}(\bar{V}^j \bar{Z}) - c \Big] \tag{3.43}$$

but since $(\bar{V}^j)^{p^m} = \bar{V}^j$, multiplying both sides by $\bar{V}^{-j}$ we get:

$$\bar{Z}^{p^m} - \bar{Z} = (\bar{V}^{-j}) \prod_{c \in \mathbb{F}_p} \Big[Tr^{\Lambda}_{\mathbb{F}_p}(\bar{V}^j \bar{Z}) - c \Big] \tag{3.44}$$

which yields the generalized version of (3.40):

$$H(\bar{Z}) = \prod_{c \in \mathbb{F}_p} gcd \Big(H(\bar{Z}), Tr^{\Lambda}_{\mathbb{F}_p}(\bar{V}^j \bar{Z}) - c \Big) \tag{3.45}$$

Finally it can be proved that if $deg(H(\bar{Z})) \geq 2$, then there exists at least one $j \in [0, \ldots, m-1]$ such that (3.45) yields a non trivial factorization of $H(\bar{Z})$.

The algorithm for root finding has been implemented in the translation unit `rootfind.c`. Further information along with proofs of many results used here can be found in [2].

3.2.5 Oil and Vinegar

In subsection 3.2.4 we learnt how to solve (3.19) in the general case. Now let's look for a way to strenghten the trapdoor Υ, while still sticking to a second degree equivalent representation for the public part. Let $\nu = \langle \nu_1, \ldots, \nu_r \rangle$ be an r dimensional vector with coefficients in $\mathbb{F}_2$, then let:

$$F_v(\bar{Z}) = \Upsilon(\bar{Z}) + \sum_{i=1}^{r} \big[\nu_i \cdot L_i(\bar{Z}) \big] = \bar{B} \tag{3.46}$$

where $L_i(\bar{Z})$ are linear polynomials.

The ν vector acts as a switch which selects a subset of the L_i to include in the summation. This method of dressing Υ by selectively adding linear terms raises the degree of the summation to quadratic, due to the products $\nu_i \cdot L_i(\bar{Z})$, but the overall degree of F_ν is still only two.

Another way to strenghten Υ is to remove equations from the public representation of (3.5) which as a side effect also reduces the output size.

These two methods are called *vinegar* and *oil* respectively, recalling the ingredients used to dress food and make it more tasty.

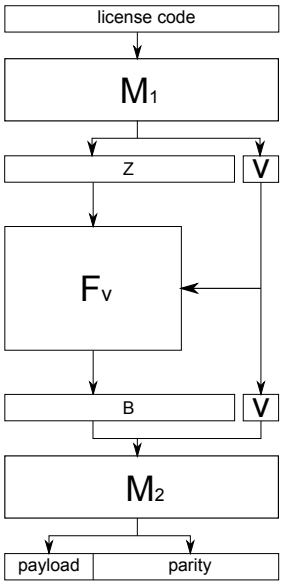

Figure 3.1: general quadratic function

Finally we can apply two linear transformations of input and output with two square matrices M_1 and M_2 of size $n + r$ (recall $n = m$ for polynomial based trapdoor functions). Figure 3.1 shows a generalized version of the complete transformation.

The overall representation with public polynomials has still the form of (3.4) with n equations in n unknowns. The license code is $n + r$ bit long. We can reverse each step for encoding the parity and the payload into the license, provided we added enough redundancy bits to afford a number of retries when $F_\nu(\bar{Z}) = \bar{B}$ has no solutions.

Figure 3.2 shows the details of the F_v function. The quadratic part is the $\Upsilon(\bar{Z})$ coefficient, while the selector along with (4 in this example) linear factors represent the other term of (3.46).

Notice that the public equations (3.4) include the full transformation (i.e. also M_1 and M_2).

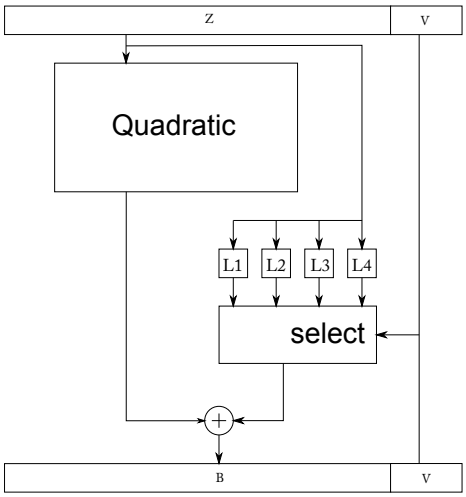

Figure 3.2: details of the $F_v(Z) = B$ core function

Chapter 4

The Quartz World

Quartz is a signature scheme based on Hidden Field Equations, more specifically on HFEV$^-$. *V* stands for *vinegar* and - for the removal of some equations. It was presented at Crypto Nessie, a European research project funded in 2000 - 2003 to identify secure cryptographic primitives. The algorithm was eventually discarded for various reasons, but it's still unbroken. You may read more about it in [8]. Quartz offers a very short signature length, only 128 bits, and the signature verification is very fast.

You might wonder why I'm writing about a signature algorithm here. For two reasons: first the trapdoor function we presented in the former section, with proper parameters, is equivalent to the Quartz core function, on which the algorithm bases its security. Second and most important, the *QRegZ* application included in this package generates license codes by signing a *license ID* 32-bit counter with this algorithm.

I won't spend much time on Quartz because there's a lot of litterature on the net you might consult for knowledge. There's not much to spend on *QRegZ* either except that a 32 bit counter plus a 128 bit signature account for a total of 160 bit length vector, which translate into a 32 characters license string. A little over the edge. The scheme is also a little different from the one presented in figure 2.1 and the decoding function is much more lengthy than the one presented for *DRegZ* , i.e.: SDDecoder.

4.1 *JRegZ*

JRegZ tries to overcome the issues of *DRegZ* related to security and *QRegZ* to key length and decoding. Also neither of the former two were equipped with a flexible payload for it accounts only the key ID and redundancy (actually this is an implementation drawback, not a structural one).

The trapdoor function used in *JRegZ* is still (3.46) and operates on n bits input, with $64 \leq n \leq 128$. The encoder however requires also n to be divisible by 5 to match with the 5 bits per symbols, so in practice we have lower and

upper bounds of 65 and 125 respectively, with n multiple of 5. The number of vinegar bits is fixed to 4, and thus the degree d_I of the irreducible polynomial defining the extension field is $d_I = n - 4$.

The degree d of (3.46) - not to be confused with d_I - plays an important role for security. The higher the better, but it also slows the root finding algorithm used for inversion. Good values for d are in the form $2^k + 1$, with $7 \leq k \leq 9$, yielding for d values in the set $\{129, 257, 513\}$. We thus defined a security level l ranging from 1 to 3 that can be set as a parameter, with: $k = l + 6$ and thus $d = 2^{l+6} + 1$.

4.1.1 C++ Class Interface

JRegZ has been designed as an expandable C++ library rather than as standalone program like the *DRegZ* and *QRegZ* , although both programs can be turned into a library with some work. The idea is that once you add flexibility with features bits, you have to drop the automatism of canned programs and write your own application which drives the encoder (you might also pass bitstring to example programs as parameters, but having a human talking to a program in binary is not exactly what I would define a user friendly interface).

Our first observation is that *DRegZ* and *JRegZ* share the same decoding scheme, because they differ only on how they build the set of equations. For this reason the steps needed for inversion are different but not the straight core function evaluation. I nevertheless added some new decoding routines called jdecoder, because the older SDDecoder wasn't able to handle variable sized payloads/bitstrings.

There exist two C++ classes to work with *JRegZ* : CLicenseCode which handles generic operations such encoding/verifying licenses, generating keypairs and setting global parameters like license size. The other class, CJRegZ, implements the JRegZ specific algorithm. CJRegZ is derived from CMVTrap as should every other class supporting algorithms which share the same decoding scheme. Just as an example there exists also a CDRegZ class, derived from CMVTrap which implements the DRegZ scheme.

A good place to start understanding how these classes interact is the static function CLicenseCode::AutoTest(). That function shows how to create a keypair, save and load it from disk, and generate/verify licenses. It also shows how to produce C language code to be compiled and linked with jdecoder routines in order to create an independent decoding application.

Notice that for portability and ease of use a global CPRNG is embedded into the baseclass CMVTrap and must be seeded before generating a real keypair through the static function CMVTrap::SeedPrng.

Bibliography

[1] Joachim von zur Gathen and Jürgen Gerhard
Modern Computer Algebra
Cambridge University Press; 2nd edition.
ISBN-10: 0521826462
ISBN-13: 978-0521641760

[2] Rudolf Lidl and Harald Niederreiter
Introduction to Finite Fields and their Applications
Cambridge University Press; 2nd edition (August 26, 1994)
ISBN-10: 0521460948
ISBN-13: 978-0521460941

[3] Robert J. McEliece
Finite Fields for Computer Scientists and Engineers
Springer; 1 edition (November 30, 1986)
ISBN-10: 0898381916
ISBN-13: 978-0898381917

[4] Jacques Patarin and Louis Goubin
Assymmetric Cryptography with S-boxes (extendend version)

[5] Jean-Charles Faugére homepage:
`http://www-calfor.lip6.fr/~jcf/`

[6] Nicolas T. Courtois, Louis Goubin and Jacques Patarin
Quartz, an assymmetric signature scheme for short signatures on PC
`http://www.minrank.org/quartz-b.pdf`
Homepage (possibly stale)
`http://www.cryptosystem.net/quartz/`

[7] Christopher Wolf
Efficient Public Key Generation for Multivariate Cryptosystems
Cryptology ePrint Archive, Report 2003/089

[8] Nessie: New European Schemes for Signatures, Integrity, and Encryption
Homepage
`http://www.cryptonessie.org`

[9] SDDecoder challenge and solution
`http://www.crackmes.de/users/gbe32241/sddecoder`