ПОЛИНОМИАЛЬНАЯ АРИФМЕТИКА И ПОЛЯ ГАЛУА

Искусство рассуждать – это искусство обманывать самого себя
Антуан де Сент-Экзюпери
«Цитадель»

ИЛИ ИНФОРМАЦИЯ, ВОСКРЕСШАЯ ИЗ ПЕПЛА II

В прошлой статье этого цикла мы говорили о том, что помехоустойчивые коды Рида-Соломона основаны на двух фундаментальных математических составляющих: полиномиальной арифметике и арифметике полей Галуа. До тех пор, пока эти вопросы не будут нами всесторонне рассмотрены, мы не сможем двигаться дальше и потому наберемся чуточку терпения, чтобы совершить решительный штурм математических вершин. После чего начнется чистое программирование, практически без примесей всяких инородных математик.

КРИС КАСПЕРСКИ

Полиномиальная арифметика

Полиномиальной арифметике посвящен шестой раздел третьего тома «Искусства программирования» Дональда Кнута, где полиному дается следующее определение: *«Формально говоря, полином над S представляет собой выражение вида: $u(x) = u_n x^n + \ldots + u_1 x + u_0$, где коэффициенты $u_n, \ldots, u_1, u_0$ – элементы некоторой алгебраической системы S, а переменная x может рассматриваться как формальный символ без определяющего значения. Будем полагать, что алгебраическая система S представляет собой коммутативное кольцо с единицей. Это означает, что S допускает операции сложения, вычитания и умножения, удовлетворяющие обычным свойствам: сложение и умножение являются ассоциативными и коммутативными бинарными операциями, определенными на S, причем умножение дистрибьютивно по отношению к сложению. Существует также единичный элемент по сложению 0 и единичный элемент по умножению 1, такие, что $a + 0 == a$ и $a^1 == a$ для всех a из S. Вычитание является обратной по отношению к сложению операцией, но о возможности деления как операции, обратной по отношению к умножению, ничего не предполагается. Полином $0*x_{n+m} + \ldots + 0*x_{n+1} + u_n x^n + \ldots + u_1 x + u_0$ рассматривается как идентичный $u_n x^n + \ldots + u_1 x + u_0$, хотя формально он отличается от него».*

Таким образом, вместо того, чтобы представлять информационное слово D, кодовое слово C и остаток от деления R в виде целых чисел (как это делалось нами ранее), мы можем связать их с соответствующими коэффициентами двоичного полинома, выполняя все последующие математические манипуляции по правилам полиномиальной арифметики. Выигрыш от такого преобразования на первый взгляд далеко не очевиден, но не будем спешить, а лучше преобразуем любое пришедшее нам в голову число (например, 69h) в двоичный полином. Запустив «Калькулятор» или любое другое подходящее приложение по вашему вкусу, переведем наше число в двоичный вид (при соответствующих навыках эту операцию можно выполнить и в уме, см. «Техника и философия хакерских атак» Криса Касперски): 69h → 1101001.

Ага, крайний правый коэффициент равен единице, затем следуют два нулевых коэффициента, потом единичный коэффициент… Короче говоря, получается следующее: $1*x^6 + 1*x^5 + 0*x^4 + 1*x^3 + 0*x^2 + 0*x + 1$. По сути говоря, битовая строка «1101001» является одной из форм записи вышеуказанного полинома – ненаглядной с точки зрения неподготовленного человека, но удобной для машинной обработки. Постойте, но если 69h уже представляет собой полином, то в чем разница между сложением полиномов 69h и 27h и сложением целых чисел 69h и 27h?! Разница несомненно есть. Как еще показал Ницше: фактов нет, а есть одни лишь интерпретации. Интерпретация же чисел и полиномов различна, и математические операции над ними выполняются по совершенно независимым правилам.

Коэффициенты в полиномиальной арифметике строго типизированы и коэффициент при x^k имеет иной тип, нежели при x^m (конечно, при том условии, что $k \neq m$).

А операции над числами различных типов категорически не допустимы! Все коэффициенты обрабатываются независимо, а возникающий при этом перенос в старший разряд (заем из старшего разряда) попросту не учитывается. Покажем это на примере сложения чисел 69h и 27h:

Листинг 1. Сложение, выполненное по правилам полиномиальной двоичной арифметики (слева) и сложение, выполненное по правилам обычной арифметики (справа).

```
 1101001  (69h)          1101001   (69h)
+0100111  (27h)         +0100111   (27h)
 -------                 -------
 1001110  (4Eh)          10010000  (90h)
```

Простейшие расчеты показывают, что сложение полиномов по модулю два дает тот же самый результат, что их вычитание, и «волшебным» образом совпадает с битовой операцией XOR. Впрочем, совпадение с XOR – чистая случайность, но вот эквивалентность сложения и вычитания заставляет заново пересматривать привычную природу вещей, вспоминая задачки из серии «у Маши было одно яблоко, Петя отнял у нее его, затем ей подарили еще одно, спрашивается: сколько всего яблок у Маши осталось? А сколько у нее было бы, если бы первое яблоко осталось не отнятым?». С точки зрения арифметики по модулю два ответ: один и ноль соответственно. Да! Не отними бы Петя у Маши яблоко, 1 + 1 == 0 и бедная Маша вообще осталась бы ни с чем. Так что, мальчики, почаще отнимайте яблоки у девушек – учите их компьютерной грамотности!

Впрочем, мы отвлеклись. Вернемся к фиктивному члену x нашего полинома и его коэффициентам. Благодаря их типизации и отсутствию взаимных связей, мы можем осуществлять обработку сколь угодно длинных чисел, просто XOR составляющие их биты на потоке. Это и есть одно из тех достоинств полиномиальной арифметики, которые не видны с первого взгляда, но благодаря которым полиномиальная арифметика стала так широко распространена.

Однако в нашем случае одной лишь полиномиальной арифметикой дело не обходится и для реализации кодера/декодера Рида-Соломона нам потребуется активная помощь со стороны полей Галуа. Что же это за поля такие, спросите вы?

Поля Галуа

В далеких шестидесятых, когда компьютеры были большими, а 20 Мб винчестеры напоминали собой стиральные машины, родилась одна из красивейших легенд о зеленом инопланетном существе, прилетевшем со звёзд и записавшем всю Британскую энциклопедию на тонкий металлический стержень нежно-серебристого цвета, который существо и увезло с собой. Сегодня, когда габариты 100 Гб жестких дисков сократились до размеров сигаретной пачки, такая плотность записи информации уже не кажется удивительной и даже вызывает улыбку. Но! Все дело в том, что инопланетное существо обладало технологией записи бесконечного количества информации на бесконечно крошечном отрезке и Британская энциклопедия была выбрана лишь

для примера. С тем же успехом инопланетянин мог скопировать содержимое всех серверов Интернета, нанеся на свой металлический стержень всего одну-единственную риску. Не верите? А зря! Переводим Британскую энциклопедию в цифровую форму, получая огромное-преогромное число. Затем – ставим впереди него запятую, преобразуя записываемую информацию в длиннющую десятичную дробь. Теперь только остается найти два числа A и B, таких, что результат деления A и B как раз и будет равен данному числу с точностью до последнего знака. Запись этих чисел на металлический стержень осуществляется нанесением риски, делящей последний на два отрезка с длинами, кратными величинам A и B соответственно. Для считывания информации достаточно всего лишь измерить длины отрезков A и B, а затем – поделить один на другой. Первый десяток чисел после запятой будет более или менее точен, ну а потом… Потом жестокая практика опустит абстрактную теорию по самые помидоры, окончательно похоронив последнюю под толстым слоем информационного мусора, возникающего из невозможности точного определения геометрических размеров объектов реального мира.

В цифровом мире дела обстоят еще хуже. Каждый программист знает, что на деление целых и вещественных чисел наложены достаточно жесткие ограничения. Помимо того, что деление весьма прожорливая в плане процессорных ресурсов операция, так она еще и математически неточная!

То есть, если c = a * b, то еще не факт, что a == c/b! Таким образом, для практической реализации кодов Рида-Соломона обычная арифметика непригодна и приходится прибегать к помощи особой математики – математики конечных групп Галуа.

Под группой здесь понимается совокупность целых чисел, последовательно пронумерованных от 0 до $2^n - 1$, например: {0, 1, 2, 3} или {00h 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h, 09h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh}. Группы, содержащие 2^n элементов, называются полями Галуа (Galois Field) и обозначаются так: $GF(2^n)$[1].

Члены групп в обязательном порядке подчиняются ассоциативному, коммутативному и дистрибьютивному законам, но обрабатываются довольно противоестественным на первый взгляд образом:

1) сумма двух любых членов группы всегда присутствует в данной группе;
2) для каждого члена «a» группы существует тождественный (identity) ему член, обычно записываемый как «e», удовлетворяющий следующему условию: a + e = e + a = a;
3) для каждого члена «a» группы, существует обратный (inverse) ему член «–a», такой, что: a + (–a) == 0.

Начнем с первого тезиса. Не кажется ли он вам бредом? Допустим, у нас есть группа {0, 1, 2, 3}. Это каким же в дупель пьяным нужно быть, чтобы при вычислении значения 2 + 3 получить число меньшее или равное 3?! Оказывается, сложение в полях Галуа осуществляется без учета переноса и сумма двух членов группы равна: c = (a + b) % 2^n, где операция «%» обозначает взятие остатка. Применительно к нашему случаю: (2 + 3) % 4 == 1. У математиков это называется «сложением по модулю 4».

Естественно, вас интересует: а применяется ли сложение по модулю на практике или используется лишь в абстрактных конструкциях теоретиков? Хороший вопрос! Сложение по модулю мы машинально выполняем десятки раз на дню, даже не задумываясь о том, что это и есть сложение без учета переноса. Вот, например, проснувшись в шесть вечера по утру, вы просидели за компьютером девять часов кряду, а потом неожиданно бросили взгляд на свои наручные часы. Какое положение занимала часовая стрелка в это время, при условии, что часы идут точно? Искомое значение со всей очевидностью представляет собой сумму 6 и 9 по модулю 12 и равно оно: (6 + 9) % 12 == 3. Вот вам наглядный пример практического использования арифметики Галуа. А теперь давайте в порядке эксперимента вычтем из числа 3 число 6… (если не догадываетесь, как это правильно сделать, – возьмите в руки часы).

Теперь самое главное: раз результат деления одного члена группы на другой, естественно, не равный нулю, член в обязательном порядке должен присутствовать в данной группе, то несмотря на то, что деление осуществляется в целых числах, оно будет точным. Точным, а не округленным! Следовательно, если c = a * b, то a == c/b. Другими словами, умножение и деление непротиворечивым образом определено для всех членов группы, конечно, за исключением невозможности деления на нуль, причем расширения разрядной сетки при умножении не происходит!

Конечно, это не совсем обычное умножение (и далеко не во всяком поле Галуа дважды два будет рано четырем), однако никто и не требует от арифметики Галуа ее соответствия «здравому смыслу» и «житейскому опыту». Главное – что она работает, причем работает хорошо. И существование жестких дисков, CD-ROM/DVD приводов – лучшее тому подтверждение, ибо все они так или иначе используют эту арифметику в своих целях.

Как уже говорилось, в вычислительной технике наибольшее распространение получили поля Галуа с основанием 2, что объясняется естественностью этих полей с точки зрения машинной обработки, двоичной по своей природе.

Для реализации кодера/декодера Рида-Соломона нам потребуются четыре базовых арифметических операции: сложение, вычитание, умножение и деление. Ниже они будут рассмотрены во всех подробностях.

Сложение и вычитание в полях Галуа

Сложение по модулю два в полях Галуа тождественно вычитанию и реализуется битовой операцией XOR. Этот вопрос мы уже обсуждали при изучении полиномиальной арифметики, поэтому не будем лишний раз повторяться, а просто приведем законченный пример программной реализации функции сложения/вычитания:

Листинг 2. Функция, реализующая сложение/вычитание в полях Галуа.

```
// функция возвращает результат сложения (вычитания)
// двух полиномов a и b по модулю 2
int gf_sum(int a, int b)
{
    return a ^ b;
}
```

Умножение в полях Галуа

Открыв учебник математики за третий класс (если мне не изменяет память), мы найдем, что умножение представляет собой многократное сложение и, коль скоро сложение в полях Галуа мы выполнять уже научились, мы имеем все основания считать, что реализация функции умножения не создаст особого труда. Так? А вот и нет! Я всегда знал, что дважды два равно четырем, до конца никогда не верил в это и, впервые столкнувшись с полями Галуа, понял, насколько был прав[2]. Выяснилось, что существуют и такие математики, где дважды два не равно четырем, а операция умножения определяется не через сложение, а совсем по-другому.

Действительно, если попытаться «обернуть» функцию gf_sum в цикл, мы получим то же самое сложение только в профиль. a * b будет равно a, если b четно, и нулю, если b нечетно. Ну и кому такое умножение нужно? Собственно, функция «настоящего» умножения Галуа настолько сложна и ресурсоемка, что для упрощения ее реализации приходится прибегнуть к временному преобразованию полиномов в индексную форму, последующему сложению индексов, выполняемому по модулю GF, и обратному преобразованию суммы индексов в полиномиальную форму.

Что такое индекс? Это – показатель степени при основании два, дающий искомый полином. Например, индекс полинома 8 равен 3 ($2^3 = 8$), а индекс полинома 2 равен 1 ($2^1 = 2$). Легко показать, что $a * b = 2^i * 2^j = 2^{(i+j)}$. В частности, $2 * 8 = 2^3 * 2^1 = 2^{(3+1)} = 2^4 = 16$. Составим следующую табличку и немного поэкспериментируем с ней:

Таблица 1. Таблица полиномов (левая колонка) и соответствующих им степеней двойки (правая колонка).

i	alpha_of[i]
001	0
002	1
004	2
008	3
016	4

До сих пор мы оперировали понятиями привычной нам арифметики, и потому добрые две трети полей таблицы остались незаполненными. В самом деле, уравнения типа $2^x = 3$ в целых числах не разрешимы и ряд индексов не соответствует никаким полиномам! Так-то оно так, но в силу того, что количество полиномов всякого поля Галуа равно количеству всевозможных индексов, мы можем определенным образом сопоставить их друг другу, закрыв глаза на то, что с точки зрения обычной математики такое действие не имеет никакого смысла. Конкретная схема сопоставления может быть любой, главное – чтобы она была внутренне непротиворечивой, то есть удовлетворяла всем правилам групп, перечисленным выше (см. «Поля Галуа»).

Естественно, поскольку от выбранной схемы сопоставления напрямую зависит и конечный результат, обе стороны (кодер и декодер Рида-Соломона) должны соблюдать определенные договоренности. Однако различные кодеры/декодеры Рида-Соломона могут использовать различные схемы сопоставления, несовместимые друг с другом.

В частности, декодер Рида-Соломона, встроенный в CD-ROM привод, выполняет умножение по следующей таблице. Встретив такую таблицу в дизассемблерном листинге исследуемой вами программы, вы сможете быстро и надежно отождествить использующие ее функции:

Таблица 2. Lock-up-таблица для GF(256). Первая слева колонка – полиномы/индексы (обычно обозначается, как i), вторая – таблица степеней примитивного полинома 2 (обычно обозначается как alpha_of), третья – индексы, соответствующие данному полиному (обычно обозначается как index_of).

i	alpha	index	i	alpha	index	i	alpha	index	i	alpha	index	i	alpha	index	i	alpha	index
000	001	-1	043	119	218	086	177	219	129	023	112	172	123	220	215	239	170
001	002	0	044	238	240	087	127	189	130	046	192	173	246	252	216	195	251
002	004	1	045	193	18	088	254	241	131	092	247	174	241	190	217	155	96
003	008	25	046	159	130	089	225	210	132	184	140	175	255	97	218	043	134
004	016	2	047	035	69	090	223	19	133	109	128	176	227	242	219	086	177
005	032	50	048	070	29	091	163	92	134	218	99	177	219	86	220	172	187
006	064	26	049	140	181	092	091	131	135	169	13	178	171	211	221	069	204
007	128	198	050	005	194	093	182	56	136	079	103	179	075	171	222	138	62
008	029	3	051	010	125	094	113	70	137	158	74	180	150	20	223	009	90
009	058	223	052	020	106	095	226	64	138	033	222	181	049	42	224	018	203
010	116	51	053	040	39	096	217	30	139	066	237	182	098	93	225	036	89
011	232	238	054	080	249	097	175	66	140	132	49	183	196	158	226	072	95
012	205	27	055	160	185	098	067	182	141	021	197	184	149	132	227	144	176
013	135	104	056	093	201	099	134	163	142	042	254	185	055	60	228	061	156
014	019	199	057	186	154	100	017	195	143	084	24	186	110	57	229	122	169
015	038	75	058	105	9	101	034	72	144	168	227	187	220	83	230	244	160
016	076	4	059	210	120	102	068	126	145	077	165	188	165	71	231	245	81
017	152	100	060	185	77	103	136	110	146	154	153	189	087	109	232	247	11
018	045	224	061	111	228	104	013	107	147	041	119	190	174	65	233	243	245
019	090	14	062	222	114	105	026	58	148	082	38	191	065	162	234	251	22
020	180	52	063	161	166	106	052	40	149	164	184	192	130	31	235	235	235
021	117	141	064	095	6	107	104	84	150	085	180	193	025	45	236	203	122
022	234	239	065	190	191	108	208	250	151	170	124	194	050	67	237	139	117
023	201	129	066	097	139	109	189	133	152	073	17	195	100	216	238	011	44
024	143	28	067	194	98	110	103	186	153	146	68	196	200	183	239	022	215
025	003	193	068	153	102	111	206	61	154	057	146	197	141	123	240	044	79
026	006	105	069	047	221	112	129	202	155	114	217	198	007	164	241	088	174
027	012	248	070	094	48	113	031	94	156	228	35	199	014	118	242	176	213
028	024	200	071	188	253	114	062	155	157	213	32	200	028	196	243	125	233
029	048	8	072	101	226	115	124	159	158	183	137	201	056	23	244	250	230
030	096	76	073	202	152	116	248	10	159	115	46	202	112	73	245	233	231
031	192	113	074	137	37	117	237	21	160	230	55	203	224	236	246	207	173
032	157	5	075	015	179	118	199	121	161	209	63	204	221	127	247	131	232
033	039	138	076	030	16	119	147	43	162	191	209	205	167	12	248	027	116
034	078	101	077	060	145	120	059	78	163	099	91	206	083	111	249	054	214
035	156	47	078	120	34	121	118	212	164	198	149	207	166	246	250	108	244
036	037	225	079	240	136	122	236	229	165	145	188	208	081	108	251	216	234
037	074	36	080	253	54	123	197	172	166	063	207	209	162	161	252	173	168
038	148	15	081	231	208	124	151	115	167	126	205	210	089	59	253	071	80
039	053	33	082	211	148	125	051	243	168	252	144	211	178	82	254	142	88
040	106	53	083	187	206	126	102	167	169	229	135	212	121	41	255	000	175
041	212	147	084	107	143	127	204	87	170	215	151	213	242	157			
042	181	142	085	214	150	128	133	7	171	179	178	214	249	85			

С помощью данной таблицы вы легко сможете осуществлять преобразование из полиномиальной формы в индексную и наоборот. Как пользоваться этой таблицей? Допустим, мы хотим умножить полиномы 69 и 96. Находим в первой колонке число 69. Ему соответствует alpha 47, запоминаем (записываем его на бумажке) и переходим к числу 96, alpha которого равен 217. Складываем 47 и 217 по модулю 256, получая в результате: (217 + 47) % 256 = 8. Теперь переводим результат произведения из индексной формы в полиномиальную: находим в первой колонке число 8 и в третьей колонке видим соответствующий ему полином: 3. (Если же мы вы-

полним обратную операцию, разделив 3 на 69 – мы получим 96, что доказывает непротиворечивость операций деления и умножения, а также всей арифметики Галуа в целом). Быстро, не правда ли, хотя местами и не совсем понятно, почему таблица составлена именно так, а не иначе? Хуже всего, что достоверность результата нельзя почувствовать «вживую», поскольку все это – абстракции чистейшей воды, что серьезно осложняет отладку программы (сложно отлаживать то, чей принцип работы до конца не понимаешь).

Впрочем, таблицу умножения не обязательно набивать с клавиатуры вручную и ее вполне можно генерировать и на лету, по ходу исполнения программы. Один из примеров реализации генератора выглядит так:

Листинг 3. Процедура генерации look-up таблицы быстрого умножения полиномов.

```
// степень RS-полинома (согласно Стандарта ECMA-130 - восемь)
#define m  8
// n=2*m-1 (длина кодового слова)
#define n  255
// количество ошибок, которые мы хотим скорректировать
#define t  1
// k = n-2*t (длина информационного слова)
#define k  253

// несократимый порождающий полином
// согласно Стандарту ECMA-130: P(x) = x^8 + x^4 + x^3 + x^2 + 1
int p[m+1]={1, 0, 1, 1, 1, 0, 0, 0, 1 };

// таблица степеней примитивного члена
int alpha_to[n+1];
// индексная таблица для быстрого умножения
int index_of[n+1];

//-----------------------------------------------------------
// генерируем look-up таблицу для быстрого умножения для GF(2^m)
// на основе несократимого порождающего полинома P©
// от p[0] до p[m].
//
// look-up таблица:
// index -> polynomial из alpha_to[] содержит j=alpha^i,
//  где alpha есть примитивный член, обычно равный 2
//  a ^ - операция возведения в степень (не XOR!);
//
//  polynomial form -> index из index_of[j=alpha^i] = i;
//
// © Simon Rockliff
//-----------------------------------------------------------

generate_gf()
{
    int i, mask;

    mask = 1; alpha_to[m] = 0;

    for (i = 0; i < m; i++)
    {
           alpha_to[i] = mask;
           index_of[alpha_to[i]] = i;

           if (p[i] != 0) alpha_to[m] ^= mask;
           mask <<= 1;
    } index_of[alpha_to[m]] = m; mask >>= 1;

    for (i = m+1; i < n; i++)
    {
           if (alpha_to[i-1] >= mask)
            alpha_to[i] = alpha_to[m] ^ ((alpha_to[i-1]^mask)<<1);
           else
            alpha_to[i] = alpha_to[i-1]<<1;

           index_of[alpha_to[i]] = i;
    } index_of[0] = -1;
}
```

Сама же функция умножения выглядит тривиально, укладываясь всего в пяток строк. В большинстве программных реализаций кодера/декодера Рида-Соломона, которые мне только доводилось видеть, операция умножения даже не выносится в отдельную процедуру, а реализуется непосредственно по месту вызова.

Листинг 4. Функция быстрого табличного умножения полиномов в полях Галуа.

```
// функция возвращает результат умножения двух полиномов
// a на b в полях Галуа
int gf_mul(int a, int b)
{
    int sum;
    // немного оптимизации не повредит
    if (a == 0 || b == 0) return 0;
    // вычисляем сумму индексов полиномов
    sum = alpha_of[a] + alpha_of[b];
    // приводим сумму к модулю GF
    if (sum >= GF-1) sum -= GF-1;
    // переводим результат в полиномиальную форму
    // и возвращаем результат
    return index_of[sum];
}
```

Деление в полях Галуа

Деление в полях Галуа осуществляется практически точно так, как и умножение, с той лишь разницей, что индексы не прибавляются, а вычитаются друг из друга. В самом деле: $a/b == 2^i/2^j == 2^{(i-j)}$. Для перевода из полиномиальной в индексную форму и наоборот может использоваться уже приводимая выше look-up таблица.

Естественно, не забывайте о том, что какими бы извращенными поля Галуа ни были, а на нуль даже в абстрактной арифметике делить нельзя и функция деления должна быть снабжена соответствующей проверкой.

Листинг 5. Функция быстрого табличного деления в полиномов в полях Галуа.

```
// функция возвращает результат деления двух полиномов
// a на b в полях Галуа, при попытке деления на ноль функция
// возвращает -1
int gf_div(int a, int b)
{
    int diff;
    // немного оптимизации не повредит
    if (a == 0) return 0;
    // на ноль делить нельзя!
    if (b == 0) return -1;
    // вычисляем разность индексов
    diff = alpha_of[a] - alpha_of[b];
    // приводим разность к модулю GF
    if (diff < 0) diff += GF-1;
    // переводим результат в полиномиальную форму
    // и возвращаем результат
    return index_of[diff];
}
```

Простейшие практические реализации

Хорошим примером воплощения кодера/декодера Рида-Соломона являются древние модели жестких дисков, разработанных в недрах фирмы IBM. Модель IBM 3370 имела простой и наглядный кодер/декодер Рида-Соломона типа (174,171) в поле Галуа GF(256). Другими словами, он оперировал 8-битными ячейками ($2^8 = 256$), и на 171 информационный байт приходилось 3 байта суммы четности, что в результате давало кодовое слово с размером 174 байт, причем, как мы увидим далее, все три байта контрольной суммы рассчитывались совершенно независимо друг от друга, поэтому фактически ко-

дер/декодер Рида-Соломона оперировал одним байтом, что значительно упрощало его архитектуру.

В современных же винчестерах кодер/декодер Рида-Соломона стал слишком навороченным, а количество контрольных байтов многократно возросло, в результате чего пришлось работать с числами противоестественных разрядностей (порядка 1408 бит и более).

Как следствие – программный код ощетинился толстым слоем дополнительных проверок, циклов и функций, чрезвычайно затрудняющих его понимание (к тому же большинство производителей железа в последнее время перешли на аппаратные кодеры/декодеры Рида-Соломона, целиком реализованные в одной микросхеме). В общем, прогресс прогрессом, а для изучения базовых принципов работы лучше использовать древние модели.

Ниже приведен фрагмент оригинальной прошивки жесткого диска IBM 3370 (только не спрашивайте: откуда он у меня взялся):

Листинг 6. Ключевой фрагмент кодера Рида-Соломона, вырванный из прошивки IBM 3370.

```
for (s0 = s1 = sm1 = i = 0; i < BLOCK_SIZE; ++i)
{
    s0  =                           s0 ^ input[i];
    s1  =         GF_mult_by_alpha[ s1 ^ input[i] ];
    sm1 =  GF_mult_by_alpha_inverse[sm1 ^ input[i] ];
};
```

Листинг 7. Ключевой фрагмент декодера Рида-Соломона, вырванный из IBM 3370.

```
// вычисляем синдром ошибки
err_i = GF_log_base_alpha[ GF_divide[s1][s0] ];
// исправляем сбойный байт
input[err_i] ^= s0;
```

Ну что, слабо нам разобраться: как он работает? Что касательно переменной s0 – с ней все предельно ясно: она хранит контрольную сумму, рассчитанную по тривиальному алгоритму. Как вы, наверное, помните, сложение в полях Галуа осуществляется логической операцией XOR, и потому: s0 += input[i].

Назначение переменной s1 выяснить сложнее, и чтобы понять суть разворачивающегося вокруг нее метаболизма, мы должны знать содержимое таблицы GF_mult_by_alpha. Несмотря на то, что по соображениям экономии бумажного пространства она здесь не приводится, ее имя говорит само за себя: содержимое s1 суммируется с очередным байтом контролируемого потока данных и умножается на так называемый примитивный член, обозначаемый как alpha, и равный двум. Другими словами: s1 = 2 * (s1 + input[i]).

Допустим, один из байтов потока данных впоследствии будет искажен (обозначим его позицию как err_i), тогда индекс искаженного байта можно определить тривиальным делением s1 на s0. Почему? Так ведь выражение s1 = 2 * (s1 + input[i]) по своей сути есть не что иное, как завуалированное умножение информационного слова на порожденный полином, динамически генерируемый на основе своего примитивного члена alpha. А контрольная сумма информационного слова, хранящаяся в переменной s0, фактически представляет собой то же самое информационное слово, только представленное в более «компактной» форме. И, как уже говорилось в предыдущей статье: если ошибка произошла в позиции x, то остаток от деления кодового слова на порожденный полином будет равен $k = 2^x$. Остается лишь по известному k вычислить x, что в данном случае осуществляется путем обращения к таблице GF_log_base_alpha, хранящей пары соответствий между k и 2^x. Коль скоро позиция сбойного байта найдена, его можно исправить путем XOR с рассчитанной контрольной суммой s0 (input[err_i] ^= s0). Конечно, сказанное справедливо только для одиночных ошибок, а искажения двух и более байт на блок данный алгоритм исправить не в силах. Собственно, для этого и присутствует третий байт контрольной суммы – sm1, защищающий декодер от «политнекорректных» попыток исправления ошибок, когда их больше одной. Если выражение s1/s0 == sm1 * s0 становится ложным, контроллер винчестера может засвидетельствовать факт наличия множественных ошибок, констатируя невозможность их исправления.

Однако, как хорошо известно, дефекты магнитной поверхности имеют тенденцию образовывать не одиночные, а групповые ошибки. И, чтобы хоть как-то компенсировать слабость корректирующего алгоритма, парни из IBM прибегли к чередованию байт. Винчестер IBM 3370 имел чередование 3:1, то есть сначала шел первый байт первого блока, за ним первый байт второго блока, за ним – первый байт третьего и только потом – второй байт первого блока. Такой трюк усиливал корректирующую способность винчестера с одной одиночной ошибки, до трех последовательно искаженных байт... Однако, если разрушению подвергались не соседние байты, то корректирующая способность вновь опускалась до значений в один искаженный байт на блок, но вероятность такого события была несравненно меньше.

Естественно, что данный алгоритм может быть реализован не только в самом жестком диске, но и вне его. Варьируя размер блоков и степень чередования, вы обеспечите себе лучшую или худшую защищенность при большей или меньшей избыточности информации. Действительно, пусть у нас есть N секторов на диске. Тогда, разбив их на блоки по 174 сектора в каждом и выделив 3 сектора для хранения контрольной суммы, мы сможем восстановить по меньшей мере N/174 секторов диска. Исходя из средней емкости диска в 100 Гб (что соответствует 209 715 200 секторам), мы сможем восстановить до 1 205 259 секторов даже при их полном физическом разрушении, затратив всего лишь 2% дискового пространства для хранения контрольных сумм. Согласитесь, что редкая «сыпка» винчестера проходит столь стремительно, чтобы корректирующих способностей кода Рида-Соломона оказалось недостаточно для ее воскрешения (конечно, если эту сыпку вовремя заметить и если коэффициент чередования выбран правильно: так, что сектора, принадлежащие одному дисковому блину, обслуживались бы разными корректирующими блоками, в противном случае при повреждении поверхности одного из блинов возникнет групповая ошибка, уже неисправимая данной программой).

А как быть, если «навернется» весь жесткий диск целиком? Наиболее разумный выход – создать массив из нескольких дисков, хранящих полезную информацию вперемешку с корректирующими кодами. Главный минус такого подхода – его неэффективность на массивах, состоящих из небольшого количества жестких дисков. Разумный минимум: четыре информационных диска и один контрольный, тогда потеря любого из информационных дисков компенсируется оставшимся в живых контрольным. Ну а потерянный контрольный диск элементарным образом заменяется на новый, с последующим пересчетом всех контрольных кодов. Правда, одновременный выход двух дисков из строя – это кранты. Массив из пятнадцати дисков, двенадцать из которых – информационные, а оставшиеся три – контрольные, намного более отказоустойчив и допускает одновременный крах двух любых дисков, а при благоприятном стечении обстоятельств – и трех.

Собственно, во всем этом ничего нового нет, и соответствующие RAID-контроллеры можно купить буквально в любом магазине. Однако… мне трудно представить себе, сколько будет стоит RAID-контроллер уровня 15 и удастся ли его вообще заставить работать (по личному опыту могу сказать, что RAID-контроллеры даже начальных уровней – вещь крайне глючная, капризная и требовательная как к железу, так и к операционному окружению). Наконец, практически все RAID-контроллеры требуют наличия абсолютно идентичных, ну или близких по своим характеристикам и/или интерфейсам дисков. А коли таковых нет?

Программный RAID, активно пропагандируемый настоящим автором, всех этих недостатков лишен. Вы можете использовать диски различной геометрии и даже различной емкости, причем никто не обязывает вас сосредоточивать их в одном месте – доступ к дискам может осуществляться и по сети, причем совершенно необязательно отводить под RAID-хранилище весь диск целиком! Вы вольны произвольным образом выделять ту или иную часть дискового пространства.

Как это можно реально использовать на практике? Первое, что приходит на ум, использовать часть емкости жестких дисков под хранение избыточной информации, помогающей восстановить их в случае аварии. Если несколько компьютеров объединить в сеть (что уже давным-давно сделано и без нас), то при относительно небольших накладных расходах мы сможем восстановить любой из жестких дисков членов сети даже при полном его разрушении лишь за счет одной избыточной информации, распределенной между остальными компьютерами. Более надежного хранилища для ваших данных нельзя и придумать! Подобная схема была реализована автором этой статьи в локальных сетях нескольких фирм и доказала свою высокую живучесть, гибкость и функциональность. Необходимость в постоянном резервировании содержимого жестких дисков автоматически отпала, что в условиях одноранговой сети с отсутствующим выделенным сервером более чем актуально! А ведь такие локальные сети – не редкость (нет, я не утверждаю, что такие сети хороши, просто я констатирую факт, что они существуют в природе и в обозримом будущем вымирать не собираются).

Единственный минус программного RAID – его невысокая производительность. В частности, поставив программный RAID на сервер, обрабатывающий тысячи запросов ежесекундно и интенсивно модифицирующий большое количество файлов, вы не выиграете ничего, но… ведь само понятие «производительность» очень относительно и при достаточно быстром процессоре кодирование/декодирование информации вполне реально осуществлять и на лету безо всяких потерь в пропускной способности!

С другой стороны, если операции чтения доминируют над операциями записи, то ставить программный RAID сам Крестный Отец велел, поскольку контроль целостности считываемой информации осуществляется на «железном» уровне самим приводом и при использовании систематического кодирования (т.е. информационные слова – отдельно, байты четности – отдельно), декодеру Рида-Соломона нет никакой нужды как-то вмешиваться в этот процесс и его помощь требуется лишь тогда, когда часть информации оказывается безнадежно разрушена, что случается прямо-таки скажем не часто. Так что, право же, не стоит перекармливать фирмы, специализирующиеся на выпуске RAID, тем более что на домашний и мелкоофисный рынок они все равно не обращают внимания.

Заключение

Вот мы и разобрались с нашим первым полноценным кодером/декодером Рида-Соломона, выполненным на базе арифметики Галуа. Мы также обсудили основные моменты организации дисковых массивов и даже соблазнились написанием соответствующего драйвера для реализации программного RAID. Все, что нам требуется – это научиться обрабатывать корректирующие коды большой разрядности, корректирующая способность которых не ограничивается одними лишь одиночными ошибками, а позволяет исправлять любое наперед выбранное количество искаженных байт.

Вот об этом мы и поговорим в следующей статье этого цикла, где будет дан законченный алгоритм работы современных кодеров/декодеров Рида-Соломона и полезные советы по их самостоятельной реализации.

[1] На самом деле, полями Галуа называют любые конечные поля, но в данном контексте мы будем говорить лишь о тех полях, количество членов которых равно 2^n.

[2] Другими словами говоря, щелкая выключателем, я знаю, что сейчас загорится свет. Но я не уверен в этом (монтер перерезал провода, лампочка перегорела и т. д.) Вот так и с математикой. Та жвачка, которой пичкают нас в школе и позже в институте, – это не математика. Это набор шаманских обрядов, который нас заставляют совершать, но который не позволяет проникнуть в самую суть – в дао математики. Может, оно и к лучшему, не знаю, но во всяком случае считаю долгом сказать, что «математика» преподаваемая в средних и высших учебных заведениях, имеет к математике не больше отношения, чем программирование к терзанию мыши в Word и установке системы Windows.