Calculating Algebraic Signatures

Thomas Schwarz, S.J.

1 Introduction

A signature is a small string calculated from a large object. The primary use of signatures is the identification of objects: equal objects have identical signatures and identical signatures of two random objects imply with high probability that the objects are equal. Algebraic signatures are signatures with algebraic properties. These algebraic properties make them useful for various tasks. We review the properties and some applications further below. Algebraic signatures consist of several components, each calculated in a finite field (Galois field). The symbols themselves need to represent the atomic components of the objects, i.e. in general they need to represent bytes, bit-strings of length 8. The byte-based nature of computing thus induces us to use GF(2^8), the Galois field with 256 elements, each of which is represented as a bit string of length 8. When working only with Unicode records, we might prefer using GF(2^16), the Galois field with 64K elements, each of which is represented as a bit string of length 16.

An algebraic signature is composed of several component signatures. Each component signature is calculated according to a scheme that has been rediscovered independently by Rabin and by Schwarz. The combination of components is unique to the algebraic signatures.

Properties of algebraic signatures include [LS04]:

- The signature of a changed object can be calculated from the old signature and a signature of the changes.
- Changing up to \( m \) characters in an object will change a \( m \)-component signature for sure. Swapping two characters over a small distance will be caught for sure.
- Algebraic signatures of parity objects can be calculated from the signatures of the data objects, if the parity objects are obtained from a linear code such as a Reed Solomon code calculated over the same field, XOR-based codes, or convolutional codes.
- If we encode an object with a stream code then we can calculate the algebraic signature of the encoded object from the un-encoded object and the signature of the random string XORed to the object in order to obtain the encoded object.

2 Galois Fields

Galois fields satisfy the usual properties of the better-known fields of the rational, complex, and real numbers. They have a zero element 0, a one element 1, and two operations, addition and multiplication. There exist a unique negative element to each element and any non-zero element has a multiplicative inverse. Addition and multiplication are commutative, associative, and satisfy the distributive laws. It is well known that there exists a Galois field with \( q \) elements for all prime powers \( q \), and that this...
Galois field is unique up to algebraic isomorphy. The field is denoted usually as \( \text{GF}(q) \). Since currently Computer Science calculations are based on bytes, we will be only interested in Galois fields whose number of elements is a power of 2, and in particular in \( \text{GF}(2^8) \) and \( \text{GF}(2^{16}) \). In these cases, the elements are almost, but not always represented as unsigned bytes or unsigned double bytes. The addition is usually defined as the bitwise XOR of the elements. The definition of the multiplication is more involved and the implementations vary.

For mathematical convenience, we identify a bit string \((a_{m-1}, a_{m-2}, \ldots, a_0) \in \text{GF}(2^m)\) with the polynomial \( \sum_{i=0}^{m} a_i t^i \) over the Galois field \( \text{GF}(2) = \{0,1\} \). The latter Galois field consists of only the zero and the one element and these facts determine its operations: \(0+0=0, \ 0+1=1, \ 1+0=1, \ 1+1=0, \ 0\cdot0=0\cdot1=0, \ 1\cdot0=0, \ 1\cdot1=1\). Addition of two such polynomials also amounts to taking the XOR of the coefficient strings. Multiplication is defined by multiplying as usual, obtaining a polynomial of degree up to \(2(m-1)\), and then reducing it modulo a generator polynomial.

Galois fields are known to contain (many) primitive elements, defined by the fact that their powers yield all the non-zero elements in the Galois field. We are interested only in multiplications defined by a primitive generator polynomial, where \( t \sim (0,\ldots,0,1,0) \) is primitive.

Assume that we have a Galois field with a primitive element \( \alpha \). If \( \xi \) is a non-zero element, then there exists a uniquely determined number \( i, \ 0 \leq i \leq m-2 \), such that \( \xi = \alpha^i \).
In this case, we call \( i = \log_\alpha(\xi) \) and \( \xi = \text{antilog}_\alpha(i) \) the logarithm of \( \xi \) and the antilogarithm of \( i \), respectively.

Algebra yields the well-known logarithmic formula in \( \text{GF}(q) \):
\[
\log_\alpha(\xi \cdot \zeta) = \log_\alpha(\xi) + \log_\alpha(\zeta) \mod q - 1
\]
This allows us to calculate products (and quotients) using logarithm and antilogarithm tables for non-zero factors:
\[
\xi \cdot \zeta = \text{antilog}_\alpha(\log_\alpha(\xi) + \log_\alpha(\zeta)) \mod q - 1
\]
\[
\xi / \zeta = \text{antilog}_\alpha(\log_\alpha(\xi) - \log_\alpha(\zeta)) \mod q - 1
\]
We can avoid the awkward and expensive modulo operations by extending the logarithm and antilogarithm tables used to have indices in a range of \([- (q-1), 2(q-1)]\), but will still have to check whether one or both of the operands are zero. Alternatively, we can implement multiplication using logarithms where we assign \(-q\) to the logarithm of zero and extend the antilogarithm table to negative indices in the range of \(-q\) to \(-1\) and by setting those antilogarithms to zero.

These are not the only type of optimizations that one can try for faster implementation of Galois field multiplication. See the MASCOTS paper by Greenan, Miller, and Schwarz.
3 Signature Definition

We recall that the component signature of the file \( F = (b_0, b_1, \ldots, b_{N-1}) \) is defined to be

\[
\text{sig}_\beta(F) = \sum_{i=0}^{N-1} b_i \beta^i
\]

where \( \beta = \alpha^i \) is a power of the primitive element \( \alpha \). On occasion, it is faster to calculate \( \text{sig}_\beta(F^\text{op}) \). Obviously,

\[
\text{sig}_\beta(F^\text{op}) = \sum_{i=0}^{N-1} b_i \beta^{N-1-i}.
\]

Since the algebraic properties of \( \text{sig}(F) \) readily translate into algebraic properties of \( \text{sig}(F^\text{op}) \), we calculate the latter whenever it seems opportune. The complete algebraic signature is then simply the concatenation

\[
\text{sig}(F) = (\text{sig}_0(F), \text{sig}_\alpha(F), \text{sig}_{\alpha^2}(F), \ldots, \text{sig}_{\alpha^{N-1}}(F))
\]

We now compare possible implementations. All of these implementations will process the file character by character. In general, we use a Horner scheme

\[
\text{sig}_\beta(F^\text{op}) = \sum_{i=0}^{N-1} b_i \beta^{N-1-i} = ((b_0 \beta + b_1) \beta + b_2) + \ldots b_{N-2} \beta + b_{N-1}
\]

This leads to very compact code presented in Figure 1. The function \( \text{mul} \) returns the product of the two parameters. On general principles, it seems better to calculate the component signatures in parallel in the body of the loop. As we will see, caching has a severe impact on performance and performance trade-offs are not obvious, so that the opposite strategy, calculating each component signature separately, might be faster.

```c
GFElem sig =0;
for(int i=0; i<N; i++) {
    sig = mul(sig, beta);
    sig ^= F[i];
}
```

Figure 1: Pseudo-code for signature component calculation.

4 Precalculated Multiplication Table

We can use an individual pre-calculated table for each component (i.e. \( \alpha^i \) ) signature. We collect the values \( \text{mul}(s, \beta) \) for all Galois field elements \( s \) and place them in a table. The scheme then simply uses one of these multiplication tables for every component with the exception of \( \alpha^0 = 1 \). We present the code in Figure 2. We assume a composite signature \( \text{sig}(F) = (\text{sig}_0(F), \text{sig}_\alpha(F), \text{sig}_{\alpha^2}(F), \text{sig}_{\alpha^3}(F)) \). (For Unicode or ASCII text, the first component signature has predictable leading bits, therefore in this case it would be better to use a signature \( \text{sig}(F) = (\text{sig}_0(F), \text{sig}_{\alpha^2}(F), \text{sig}_{\alpha^3}(F), \text{sig}_{\alpha^4}(F)) \).) The code uses the tables
mul1, mul2, and mul3 to implement multiplication with $\Box$, $\Box^2$, and $\Box^3$ respectively.

Figure 2: Multiplication Table Implementation for a four-component signature.

5 Single Multiplication Table

Even if we have more than one component signature, it might be advantageous to use a single table for multiplication by $\beta$. The multiplication by larger powers of $\beta$ is then performed by multiple evaluations of the table.

6 Split Tables

Since L1 and even more so L2 cache misses are expensive, we can try to use smaller tables. If we use single table for multiplication by $\beta$, then the table size is $2^f$. We can replace this table by two smaller tables, called leftTable and rightTable. Both tables have size $2^{f/2}$ so that the total space occupied is $2^{f/2+1}$. When we multiply a Galois field element $x$ with $\beta$, we split $x$ into the right and the left portion left and right as follows. left($x$) contains the leading $f/2$ bits of $x$ and right($x$) contains the trailing $f/2$ bits. We calculate left($x$) by shifting $x$ by $f/2$ bits to the right whereas we calculate right($x$) by bitwise anding with a mask consisting of $f/2$ one-bits. Then leftTable[$y$] is defined to be $y \cdot 2^{f/2} \cdot \beta$ and rightTable[$z$] is $z \cdot \beta$. Accordingly,

$$x \cdot \beta = \text{leftTable[left(x)]} + \text{rightTable[right(x)]}$$

A single multiplication by $\beta$ now costs two table lookups, an XOR-operation, a shift, and anding with a mask instead of a single table look-up, but since the total amount of space needed for storing the table is reduced by a factor of $2^{f/2-1}$, we can expect the resulting tables to fit into cache.

Figure 3: Pseudo-code for split table multiplication for the Galois field with $2^{16}$ elements.

Even though we can extend this method to more than two tables, we only implemented it with two tables calling it the Split Table method.

7 Logarithm-Antilogarithm Table

We recall that we can multiply Galois field elements using logarithm and antilogarithm table. We use the algorithm presented in pseudo-code format in Figure 4. Here, the
logarithm table has $2^f$ entries, whereas the antilogarithm table needs to accommodate the sum of two logarithms and hence has to have size $2^{f+1}$. Alternatively, we can take the remainder of the sum of logarithms modulo $2^f-1$. An observation of A. Broder’s avoids the overhead of checking for zero-operands. We set log[0] to $-2^{f+1}$ and expands the antilog tables to negative indices as far as $2(-2^{f+1})$, setting these values to be zero. As a result, one or two zero operands correctly give the result zero.

![Figure 4: Pseudo-code for Galois field multiplication using a logarithm and antilogarithm table.](image)

When we calculate our signatures, we need not use a fully implemented version of the Galois field signature, since we only multiply with certain elements $\beta^i$, $i = 1, 2, \ldots, m-1$. The logarithms of these elements are $1, 2, \ldots, m-1$, respectively. We only have to check for one operand to be zero, but we can now use Broder’s insight with much lower costs. We set log[0] to $-m$ and extend the antilogarithm table to negative indices with values antilog[−m+1] = antilog[−m+2] = ... = antilog[−1] = 0. At the other end, the largest entry into the antilogarithm table is $(2^f-2) + (m-1)$. Since smaller tables fit better in the L1 and L2 caches and lead to less conflict, these optimizations can speed up the code execution.

The method of explicit multiplication then calculates the signature with the Horner scheme using logarithm and antilogarithm, but checking with an if-statement that neither operand is zero. Explicit Fast Multiplication uses logarithm tables with negative indices to avoid the if-statement.

### 8 Multiplication by Shifting and Broder’s Method

At first glance, it would seem that we need Galois field operations to implement these calculations. These would cost us one table look-up for the logarithm of $b_i$ and an antilogarithm, but by assuming that the file contains already the logarithms we could save one table look-up. In addition to the antilogarithm lookup, we would need to calculate the logarithm of $\beta$ and perform an XOR.

However, there is a considerably faster way using the definition of Galois field elements as polynomials modulo a generating polynomial and the operations deriving from this. We recall that in this presentation, all elements of the Galois fields are polynomials in an unknown $t$ of degree up to $f-1$ and with coefficients in \{0,1\}. We perform addition, subtraction, and multiplication by multiplying the polynomials as polynomials and then taking the remainder modulo $g(t)$, with a generator polynomial $g$ that has degree $f$ and is irreducible (i.e. not the product of two other polynomials of degree greater than zero.) If the generator polynomial $g$ is primitive, then the element $t$ is a primitive element, so that all powers of $t$ modulo $g$ make up all non-zero polynomials of degree $\leq f-1$. We assume that this is the case and use $t = \alpha$ as our primitive element.

For our implementation, we need to represent the polynomials as bit strings. We write polynomials starting with the highest degree addend and then encode it by simply taking the coefficients. For example, assume that our Galois field elements are bytes. Then the
The generator polynomial has degree 8 and a Galois field element as a polynomial has the form \( a_7t^7 + a_6t^6 + a_5t^5 + a_4t^4 + a_3t^3 + a_2t^2 + a_1t + a_0 \). This Galois field element as a bit string is \((a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)\). The one element then is \((0000 0001)\), the null element is \((0000 0000)\), the primitive element is \(\alpha = (0000 0010)\), and the addition is – as always – the XOR operation. Multiplying by \(\alpha\) is then simply multiplying by \(t\) modulo \(g\). We can achieve this multiplication in a simple way. Multiplying a polynomial by \(t\) means a left-shift by one of polynomial. If the result has an addend of degree 8, then we reduce modulo \(g\) by adding (i.e. XORing \(g\) to it). Otherwise, the result is already reduced. For example, assume \(g = (1 0001 1101)\), that is, \(g(t) = t^8 + t^4 + t^3 + t^2 + t + 1\). Let’s multiply \(f = (0110 0110)\) by \(t\) several times. First, we shift \(f\) to the left by one bit and obtain \((1100 1100)\). This is the result. For the second multiplication by \(t\) we also shift \((1100 1100)\) to the left, to obtain \((1 10101000)\). The leading 1 is an overhang. We therefore reduce modulo \(g\) by adding (XORing) \(g\) to the result. This gives \((0100 0010)\) or \((1000 0101)\). Programmatically, we shift \(f\) to the left and then AND with a carry mask \((1 0000 0000)\) to find out whether the result of the shift has a bit in position 8. If the test is positive, we XOR with \(g\).

We now implement the \(\alpha\) component signature by starting with the end of the file. Define \(s_r\) to be \(\text{sig}_\alpha(b_{N-r}, b_{N-r+1}, b_{N-r+2}, b_{N-r+3})\). Then \(s_N = \text{sig}_\alpha(F)\). Obviously, \(s_1 = b_{N-1}\). Inductively, we have

\[
s_r = b_{N-r} + \alpha \cdot s_{r-1}.
\]

Using this formula, we can inductively calculate the signature with one XOR operation, one AND operation, one if-statement execution, and on average one-half XOR operations.

A simple method by Broder reduces the average number of operations considerably. In this method, we left-shift several times before we reduce. As an example, assume that the file consists of the elements

\[
(01010101, 11111111, 00001111, 01010101)
\]

After processing the last element, our partial signature is \(0101 0101\). To process the next symbol, we left-shift \(0101 0101\) to obtain \(1010 1010\) and XOR \(0000 1111\) to it. The result is \(0101 0010\). To process symbol \(1111 1111\), we left-shift \(0101 0100\) to obtain \(0101 0010\) and XOR \(1111 1111\) to it with the result \(0101 0011\). The final element is \(0101 0011\), we left shift \(0101 0101\) to obtain \(0110 0110\) and XOR the symbol to it with result \(0110 0111\). Only at this point to we reduce mod \(g\). Essentially, we write \(0110 0111\) as \(0110 0000\oplus 0011 1111\), and use a table to determine what \(0110 0000\) modulo \(g\) is. We call \(0110\) an overhang.

Populating the overhang table is simple. We have \(g \cdot t = 10 0011 1010\) so that \(g \cdot (t+1) = 11 0010 0111\). Therefore, to remove the overhang \(011\), we XOR with \(0010 0111\).

Therefore, our signature is \(011 0011 1111 \rightarrow 0010 0111 \oplus 0011 1111 = 0001 1000\).

If we calculate the \(\alpha^2\) component signature, we multiply effectively twice with \(t\) in every recursive step. Thus, it now takes only 2 multiplications to accumulate an overhead of 4 bits.

Assume that we have a signature consisting of four components. Assume that we have an overhang reduction table of size \(4\text{K}\), i.e. for overheads of up to 12 bits. The 1-component signature only uses XOR, therefore it does not use the table at all. The \(\alpha\)-component uses the table once every 12 symbols processed, the \(\alpha^2\)-component uses the table once every 6
symbols processed, and the $\alpha^3$-component uses the table once for every 4 symbols processed. Other than the table look-ups, each processing of a symbol costs us a left-shift and an XOR. Notice that these numbers are independent of the symbol size. In addition, we need to generate the overhang reduction table, but this can be done very fast and is done once per class call.

Overall, using Broder’s mechanism of overhang reduction seems to speed up greatly signature calculation. Bringing in the file from disk or memory into the CPU for processing is an important component of the processing time.