Comments on OASIS SAM TSS v1.0, 08 March 2022
Mike Markowitz
Information Security Corp.
July 6-9, 2023

Let me start by apologizing for jumping into this rather late in the process and for possibly revisiting stale issues. I hope that this pedantic diatribe, from which text may be freely copied, will contribute to a more precise reframing of the spec – assuming, of course, that this is still an active project and someone out there wants a corrected v2.0.

Straightforward nits, corrections, and suggestions:

- Section 2.1.1: {nit} the definition of GF_POW is somewhat sloppy because, however you interpret the ellipsis, the smallest power that can be computed by the formula, as given, is GF_MUL(X,X) = X^2; even "GF_POW(X, i) = GF_MUL(X, GF_MUL(X, … , GF_MUL(X, GF_MUL(X,1))…)), with exactly i calls to GF_MUL, if i>0 and 1 if i=0," would be more accurate.
- Section 2.2.1: It might help to identify S, at least in passing, as the matrix of random share-generating polynomials that must be evaluated on share ids (in 1st column) to produce shares, and that F is precisely that evaluation – in much the same way I is referred to as an "interpolation function" in 2.2.2.
- {nit} "pseudo code" should probably be "pseudocode" (unless this is a "which side of the pond you're on" issue)
- CheckErrors: Since share[i] is a single byte identifier, what does "if share_id[i] not unique" really mean inside a loop on i? There should be pseudocode for something that captures this C++: ("length as vector doesn't agree with its cardinality as a set"):

```
std::vector<byte> share_ids;
std::set<byte> setOfNumbers(share_ids.begin(), share_ids.end());
if (setOfNumbers.size() != share_ids.size())
        throw 7; // "share ID values are not unique";
```

- CalculateShares: "S[i][0] = secret[i]" should be moved out of the inner loop on j.
- Section 2.2.2: L(i)(U) is introduced twice, and in its definition "GF_SUM" should be "GF_ADD." It could also be described as something like the "value of the i-th Lagrange polynomial."
- "I (U, V) = GF_SUM(X[i], m)" should be "I (U, V) = GF_SUM(X, m)" – GF_SUM takes a vector as first argument, not a byte – and, for a similar reason, GF_PROD in the definition of X[i] should be GF_MUL."
- In *"share*[*i*] = a vector of shares (*i* = 0 … *m*)..." the range should be "(*i* = 0 … *m-1*)," which would than agree with the ranges in CheckErrors and RecombineShares.
- "The *pseudo code* [sic] algorithm for recombining the ***n*** shares…" should be "The pseudocode for recombining the ***m*** shares…"
- CheckErrors: there's no need to compare the length of *share[0]* with itself, so the range here should be "*i = 1 to m – 1*."
- The S matrix constructed in the pseudocode of 2.2.1 for CalculateShares is the transpose of the S matrix of D.1.2, possibly causing confusion. (It did, in fact, confuse me for a time, when I skipped over the definitions and tried to reconcile the computations that followed in those two sections.) I suggest transposing D.1.2, simply using a different symbol for S in that section, or adding a note calling out the change of notation. What's amusing is that the code for RecombineShares amounts to constructing the S matrix in the D.1.2 orientation.

- Section 2.3: “the following tables apply,” “The following EXP and LOG tables are used for each defined polynomial,” and “Note: tables are defined such that…” should be replaced by precise definitions: “Each pair of tables is defined as the powers and discrete logs of a suitable field element (a specific “*primitive polynomial*”) in each representation” (see below for those elements). That they are inverse functions (away from 0) is worth mentioning but it is NOT the basis for their definition – as one of the two equally likely ways of parsing the third of these sentences would imply.
- Section 2.4: errors in test vectors – the pseudocode in 2.2.1 correctly says R is of length (m-1)L, but the test vectors here provide nL bytes for each R. The random vectors in E.2.1 are of the correct length, and I can validate all those tests.
- Section D.1.1: “Thus, P contains the *follow* [sic] elements”
- App. E. "…for both the *polynomial* and the *polynomial* [sic]" should be something like "in both of our chosen representations."
- The titles of sections E.*.1, E.2 (P011D is missing!), and E.*.2 could/should be better! Isn’t the look-up table approach also "GF(256)-based?" Perhaps, “Implementation using look-up tables,” "Using the table of powers of a primitive element," or even “Using the cyclic structure of the multiplicative group” juxtaposed with "Using polynomial arithmetic" would work – but, yeah, they're all ugly choices!

More substantive comments as well as more nits, that (may) require discussion:

- Section 2.1 (until otherwise noted): {nit} “[TSS]...provides a mechanism to be able to…” is awkward; perhaps: “[TSS] …allows us to…?”
- “…referred to as either *k*-of-*n* or *m*-of-*n* where *k* or *m…”* I think it is entirely permissible to change the “k” to “m” throughout the leading quote and eliminate the two back-reference “k” alternatives as they add nothing but confusion to the presentation. If anyone feels squeamish about this, an editorial remark could be added about the minor change.

- "Shamir’s scheme is based on *polynomial* interpolation. For interoperable systems *the* [sic] *polynomial* must be specified. The maximum number of shares or splits is determined by the specific polynomial."

"*Polynomial interpolation*" in the **first sentence**, refers to the process of finding the unique polynomial, f(x), of *degree m-1 over GF(2^8)* whose graph {(x,y) ∈ GF(2^8) x GF(2^8) | y=f(x)} passes through the m points (share_id[i], share[i]), i=0,…,m-1, corresponding to a selection of m of the n shares; that such an f(x) exists is guaranteed by construction of the shares.

In the **second sentence**, "*polynomial*" refers to an *irreducible* polynomial p of *degree 8 over GF(2)* whose maximal ideal <p> in the polynomial ring GF(2)[X] leads to an identification (“*representation*”) of GF(2^8) with the quotient field GF(2)[X]/<p>. It thus provides a concrete realization of the addition and multiplication operations in GF(2^8) (as those induced by grade-school polynomial arithmetic performed modulo p), and by natural extension, the GF(2^8)[X] arithmetic used throughout the paper. Use of the definite article with “polynomial” is inappropriate as there is *no proper antecedent* to which “the polynomial” could possibly refer, and it is likely to mislead the uninitiated into thinking it somehow refers to the same word in the preceding sentence. It doesn’t.

The claim in the **third sentence** is incorrect: the maximum number of shares is determined by the choice of field, not its representation. In TSS over GF(2^8), a random polynomial S ∈ GF(2^8)[X] selected for share

production can only be evaluated on the 255 non-zero elements of GF(2^8) (called “share ids” in the draft), thereby yielding at most 255 shares. (We take S to be of degree m-1 to obtain a threshold value of m. One byte, s ∈ GF(2^8), of a possibly longer secret is inserted into S as its constant term, ensuring that S(0)=s, and we choose at random the remaining m-1 higher order coefficients.) The choice of GF(2^8)[X] for share polynomials limits us to 8-bit secrets and dictates that we regard a long secret as a sequence of bytes, each of which is split into shares independently – a process that should be made more explicit in the text. That the bound n <= 255 is independent of the representation of GF(2^8), *i.e.*, of the choice of p, is correctly captured in the CheckErrors pseudocode of section 2.2.1.

I feel that a mathematically precise overview of the Shamir construction – couched in intuitive, geometric language that, at a minimum, describes how the shares are split and what *polynomial interpolation* does for us – deserves to be presented in the introduction. A reordering of my sentences in the preceding three paragraphs (with judicious edits) would hopefully be seen as an improvement on the current text.

- "In this specification, we select irreducible polynomials for interpolation over a finite field…"

Some may regard this as nit-picking, but again, the polynomials referred to in this paragraph are chosen for the purpose of representing GF(2^8) as two particular quotients of the ring GF(2)[X]. You do utilize these representations to compute shares and recombine them – after all, they provide the concrete means of performing GF(2^8)[X] arithmetic that is required for interoperability – but strictly speaking you're *not* selecting these polynomials "*for interpolation,*" interpreted as “*for the purpose of interpolation*.” {The existing text is much like saying "I’m going to paint my car green to go shopping," when shopping is not all you intend to do with the repainted car.}

- “…over *a* [sic] finite field (Galois field) of *the* [sic] order 2^8”
- “referred to as GF(2^8) or GF(256)”
- “*Within* [sic] the finite field,”

There’s only *one* field of order 2^8 (up to isomorphism), so the article “*a*” should be “*the*.” “*of order* 2^8” is preferred. {nit} The GF(2^8) notation better captures its common *definition* as *the* extension of GF(2) of degree 8, but unfortunately it is not used in the sequel; I suggest making the change. “*Within*” is completely inappropriate here as the polynomials about to be introduced live in GF(2)[X], not GF(2^8)!

- "The first polynomial (denoted throughout the rest of this document as 011B…"

I found this promise of a consistent use of "011B" and "011D" to be frequently broken, the polynomials being generally referred to as "polynomials 1 and 2," and explicitly written down in App. E. I think that use of "P011B" and "P011D" for both the polynomials *and* their associated representations would be better throughout. The tags 011B and 011D should also be attached to the LOG/EXP tables (as they are to the test vectors), if only to signal the dependence of those tables on the choice of irreducible (even if the dependence on choice of generator is suppressed; see following comments for more on that point).

- Sections 2.1.1 introduces the reader to the EXP and LOG tables but omits a clear explanation of their meaning. Section 2.3 makes matters worse (see below).
- “***The*** exponential function” only makes sense if a particular generator of the multiplicative group of non-zero elements of GF(2^8) is specified; otherwise, there are *many* exponential functions.

An *old mathematician* will recognize that EXP011B[0x01] = 0x03 means we're using the (residue class of the) primitive polynomial x+1 (mod P011B) as generator of the multiplicative group of GF(2)[X]/<P011B>; EXP011B captures its successive powers encoded as bytes; and LOG011B is the inverse *discrete log* function. Similarly, EXP011D[0x01] = 0x02 means we've taken x as generator in the P011D representation.

But what is a *young engineer* to make of the existing text, given that a description of the actual role of P011B and P011D is never made explicit, and EXP/LOG are not precisely defined?

I guess what I'm objecting to here is a certain lack of mathematical rigor in the document; it could be much better! I suppose one could take the position that this is an engineering document, not a math paper. But I'd counter by holding up IEEE P1363 2000 as an excellent model of a mathematically rigorous engineering document! They do exist and, I believe, they are worth trying to produce.

- "All bytes in the threshold secret sharing algorithm are interpreted as finite field elements..."

What this is trying to say is that there is an obvious correspondence between bytes $[b_7b_6\ldots b_0]$ and elements of GF(2)[X]/<p>, *i.e.*, residue classes of polynomials $b_7x^7+b_6x^6+\ldots+b_1x+b_0 \in GF(2)[X]$ modulo the chosen irreducible p. In both representations, *bytes are regarded as compact encodings of polynomial residue classes*.

The following might be a suitable introduction to section 2.3:

Arithmetic operations in GF(2)[X]/<p> can be performed naively using the polynomial arithmetic taught in grade school with the results reduced modulo p, but computations in the multiplicative group (*i.e.*, multiplication, inversion, division, and powers) can be carried out much more efficiently using a look-up table of the powers of a primitive element and another table of its discrete logs.

Once a representation is chosen primitive polynomials are abundant and easy to find; the corresponding EXP/LOG tables are easily generated; and all provide equally efficient multiplication operations with identical results. However, it's natural to look for the "smallest" primitive when we have an obvious (and canonical) ordering of our field. In the P011D representation, the smallest non-constant polynomial x = [x02] is primitive and so provides good EXP/LOG tables. However, x = [x02] is *not* primitive in the P011B representation; it has order 51:

$$\text{LOG011B[0x02]} = 0x19 = 25, \text{ so } x^{51} = ((x+1)^{25})^{51} = (x+1)^{255*5} = 1.$$

This is why we take x+1 = [0x03] as generator for the EXP/LOG tables in that representation: it's primitive and the next larger element in the lexicographic ordering of bytes/field elements.

I think it would be helpful to insert an illustrative example motivating the use of the EXP/LOG tables (thereby explaining the two different approaches to $GF(2^8)[X]$ arithmetic presented in App. E). Something along these lines, though perhaps not so folksy?

> {Here, I'll assume that the correspondence between bytes $[b_7b_6\ldots b_0]$ and field elements $v = [b_7x^7+b_6x^6+\ldots+b_0] \in GF(2)[X]/<P011B>$ has been explicitly established.}

How do we multiply two bytes, say u = [0x27] and v = [0xb5], in our P011B representation of $GF(2^8)$? Well, we could write u as the bit vector [00100111] and immediately read off the corresponding polynomial: $u = x^5+x^2+x+1$. Similarly, $v = x^7+x^5+x^4+x^2+1$. So, we could (rather naively; see E.1.2) compute u*v mod P011B by applying distributivity and reducing the resulting polynomial of degree 12 to one of degree at most 7 by (repeatedly!) using the substitution $x^8 = x^4+x^3+x+1$ (since $x^8+x^4+x^3+x+1 = 0$ in the P011B representation and we're in characteristic 2). Tedious and inefficient? You betcha!

For greater efficiency, the following alternative approach is commonly employed (see E.1.1):

{I presume this is one of the motivations for including E.*.1/E.*.2, though I've only skimmed the Cryptol and not really tried deciphering it. So, after properly defining EXP011B as the table of powers of x+1 in the (cyclic) multiplicative group of non-zero elements of GF(2)[X]/<P011B>, and LOG as the table of discrete logs with respect to that generator…}

Recall that our table EXP011B contains pre-computed powers of the primitive polynomial g = x+1 = [0x03], those powers exhaust the non-zero elements of our field, and $g^{255}=1$ (check: EXP011B[0xff] = 0x01). Since u = [0x27] = EXP011B[0x6a] – equivalently, LOG001B[0x27] = 0x6a – we have $u = g^{0x6a}$. Similarly, LOG001B[0xb5] = 0x60 implies $v = g^{0x60}$. Consequently,

$$u*v = g^{0x6a} * g^{0x60} = g^{0x6a + 0x60} = g^{0xca} \text{ mod P011B.}$$

(Here, since $g^{255}=1$, we generally add bytes in the exponent modulo 255, though the reduction is not necessary in our example). Putting this together, we find that:

$$[0x27] * [0xb5] = EXP011B[0xca] = [0xcf].$$

Thus, with a single-byte addition, a possible single-byte modular reduction, and three table lookups, we've managed to compute:

$$(x^5+x^2+x+1) * (x^7+x^5+x^4+x^2+1) = x^7+x^6+x^3+x^2+x+1 \mod (x^8+x^4+x^3+x+1)$$

{Hopefully this is correct!}

Repeating this computation in the P011D representation, using LOG011D and EXP011D, and getting a different value for the product u*v clearly illustrates the fact that agreement on the choice of representation is crucial for interoperability. {Explanation: [0x27] in the P011B representation is x^5+x^2+x+1 *mod P011B*, while [0x27] in the P011D representation is x^5+x^2+x+1 *mod P011D*, and these residue classes are not the "same;" since they live in different sets, they can't even be compared.}

There are infinitely many representations of any finite field. This standard specifies use of the representations of $GF(2^8)$ defined by P011B [AES] and P011D [BSAFE] to foster interoperability, as well as to leverage existing implementations: the arithmetic operations required to perform TSS in either of these representations are already widely available in both hardware and software.