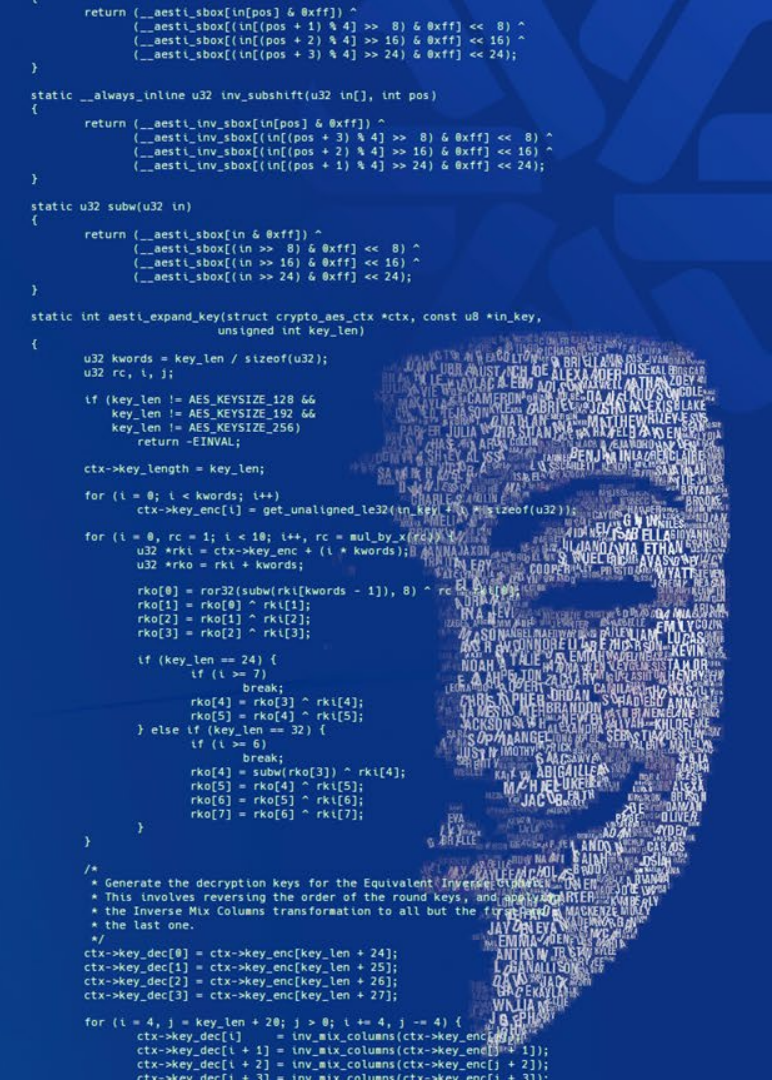
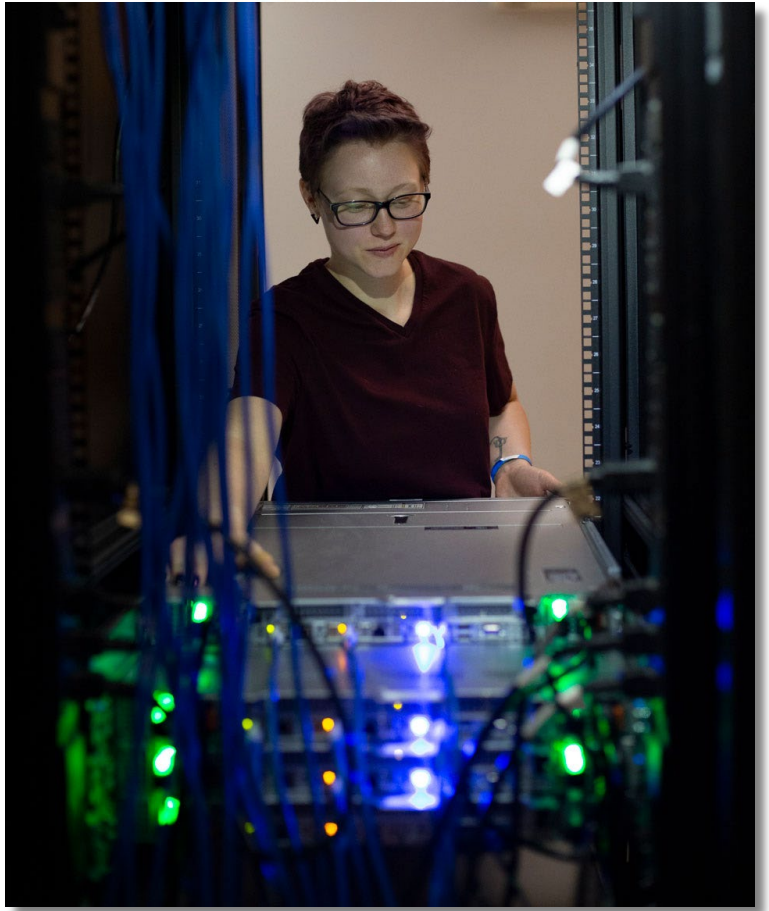
PimaCommunityCollege

Center of Excellence in Information Technology and Cybersecurity



Security Operations Center

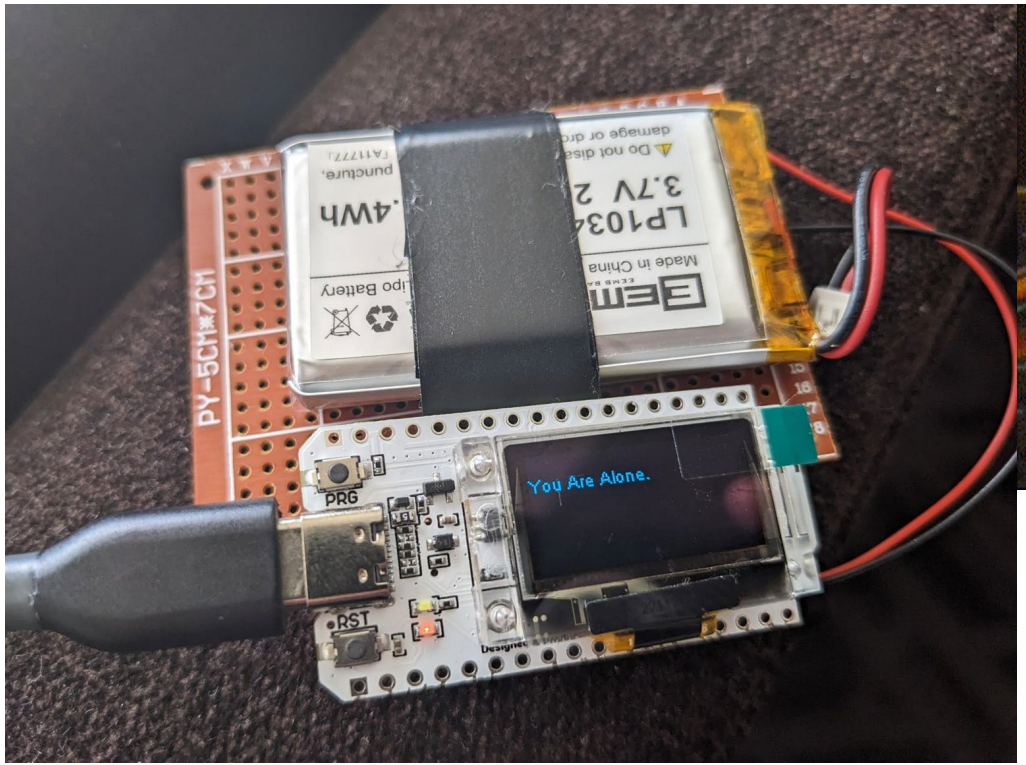


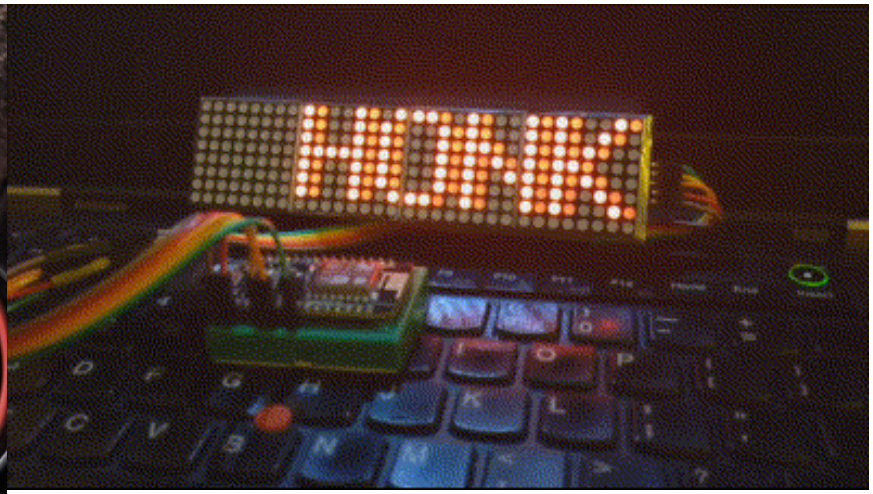
Student Managed Data Center

Arizona Cyber Warfare Range

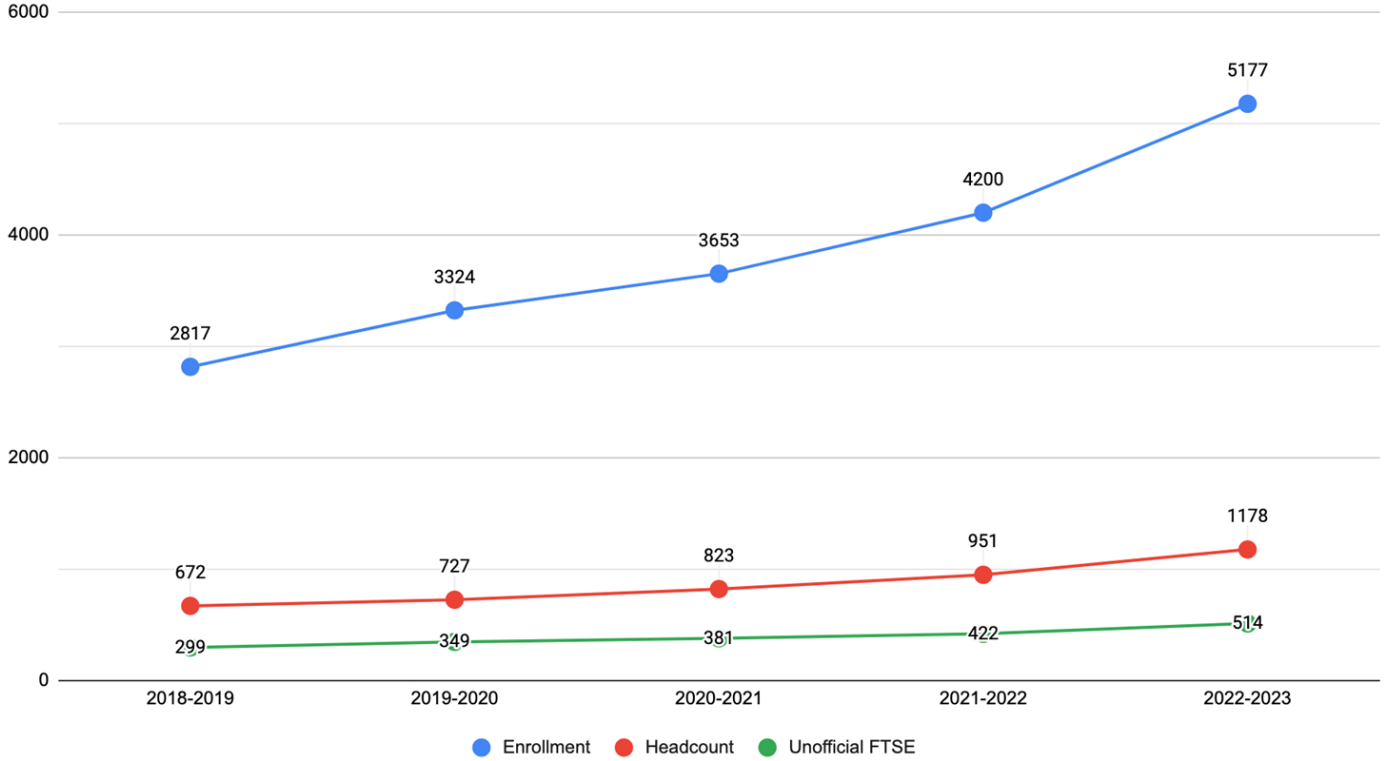
azcwr.org

Community Projects

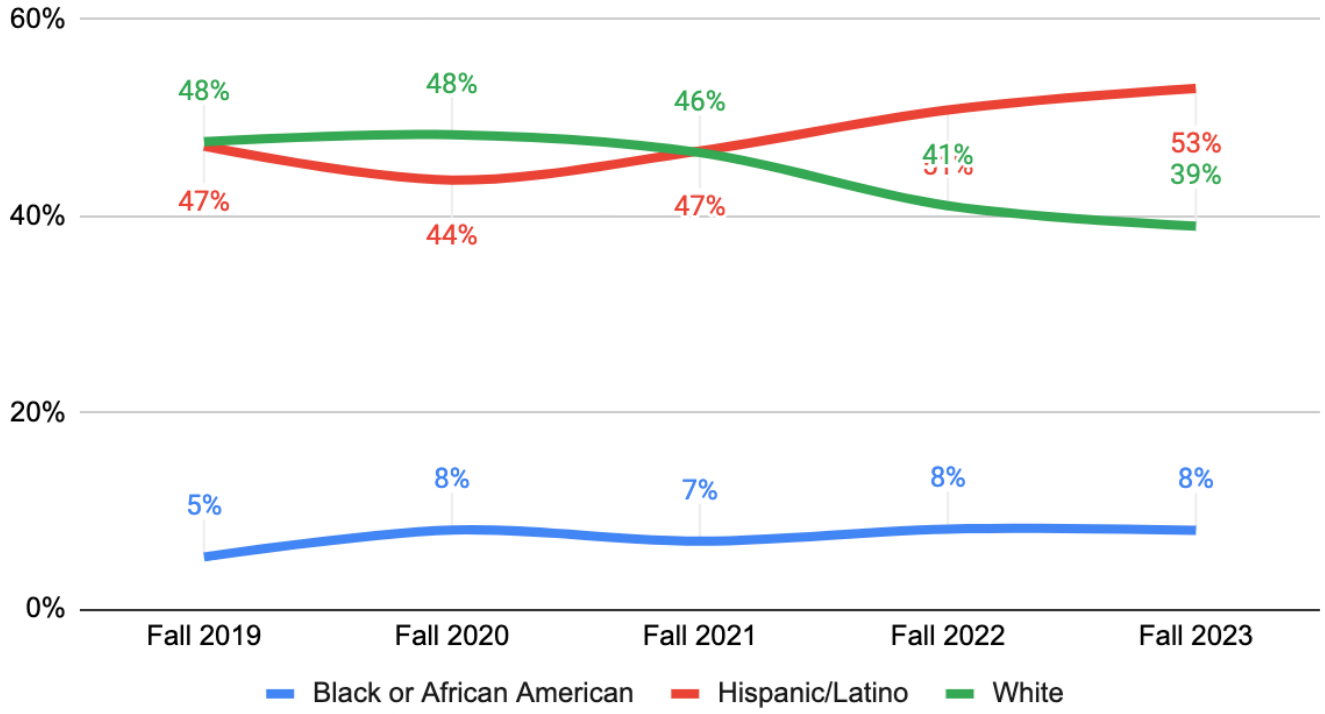




Enrollment, Headcount and Annualized Unofficial FTSE Trend



Demographic Change



PimaCommunityCollege

```
        return (__aesti_sbox[in[pos] & 0xff]) ^
               (__aesti_sbox[(in[(pos + 1) % 4] >>  8) & 0xff] <<  8) ^
               (__aesti_sbox[(in[(pos + 2) % 4] >> 16) & 0xff] << 16) ^
               (__aesti_sbox[(in[(pos + 3) % 4] >> 24) & 0xff] << 24);
}

static __always_inline u32 inv_subshift(u32 in[], int pos)
{
        return (__aesti_inv_sbox[in[pos] & 0xff]) ^
               (__aesti_inv_sbox[(in[(pos + 3) % 4] >>  8) & 0xff] <<  8) ^
               (__aesti_inv_sbox[(in[(pos + 2) % 4] >> 16) & 0xff] << 16) ^
               (__aesti_inv_sbox[(in[(pos + 1) % 4] >> 24) & 0xff] << 24);
}

static u32 subw(u32 in)
{
        return (__aesti_sbox[in & 0xff]) ^
               (__aesti_sbox[(in >>  8) & 0xff] <<  8) ^
               (__aesti_sbox[(in >> 16) & 0xff] << 16) ^
               (__aesti_sbox[(in >> 24) & 0xff] << 24);
}

static int aesti_expand_key(struct crypto_aes_ctx *ctx, const u8 *in_key,
                            unsigned int key_len)
{
        u32 kwords = key_len / sizeof(u32);
        u32 rc, i, j;

        if (key_len != AES_KEYSIZE_128 &&
            key_len != AES_KEYSIZE_192 &&
            key_len != AES_KEYSIZE_256)
                return -EINVAL;

        ctx->key_length = key_len;

        for (i = 0; i < kwords; i++)
                ctx->key_enc[i] = get_unaligned_le32(in_key + i * sizeof(u32));

        for (i = 0, rc = 1; i < 10; i++, rc = mul_by_x(rc)) {
                u32 *rki = ctx->key_enc + (i * kwords);
                u32 *rko = rki + kwords;

                rko[0] = ror32(subw(rki[kwords - 1]), 8) ^ rc ^ rki[0];
                rko[1] = rko[0] ^ rki[1];
                rko[2] = rko[1] ^ rki[2];
                rko[3] = rko[2] ^ rki[3];

                if (key_len == 24) {
                        if (i >= 7)
                                break;
                        rko[4] = rko[3] ^ rki[4];
                        rko[5] = rko[4] ^ rki[5];
                } else if (key_len == 32) {
                        if (i >= 6)
                                break;
                        rko[4] = subw(rko[3]) ^ rki[4];
                        rko[5] = rko[4] ^ rki[5];
                        rko[6] = rko[5] ^ rki[6];
                        rko[7] = rko[6] ^ rki[7];
                }
        }

        /*
         * Generate the decryption keys for the Equivalent Inverse Cipher.
         * This involves reversing the order of the round keys, and applying
         * the Inverse Mix Columns transformation to all but the first and
         * the last one.
         */
        ctx->key_dec[0] = ctx->key_enc[key_len + 24];
        ctx->key_dec[1] = ctx->key_enc[key_len + 25];
        ctx->key_dec[2] = ctx->key_enc[key_len + 26];
        ctx->key_dec[3] = ctx->key_enc[key_len + 27];

        for (i = 4, j = key_len + 20; j > 0; i += 4, j -= 4) {
                ctx->key_dec[i]     = inv_mix_columns(ctx->key_enc[j]);
                ctx->key_dec[i + 1] = inv_mix_columns(ctx->key_enc[j + 1]);
                ctx->key_dec[i + 2] = inv_mix_columns(ctx->key_enc[j + 2]);
                ctx->key_dec[i + 3] = inv_mix_columns(ctx->key_enc[j + 3]);
```