Automatic techniques for identification of cryptographic code

Hadrien Cornier, Redouane Dziri, Arthur Herbout, Corentin Llorca, Arnaud Stiegler
George Argyros, Michael McDougall
Eleni Drinea

Data Science Capstone Project with Amazon

Detecting cryptography in projects

There is a variety of cryptography libraries that implement secure cryptography algorithms such as OpenSSL. However, some open-source projects implements their own cryptography code that can potentially be insecure. Detecting those custom implementations is of paramount importance for using those projects.

Cryptography code uses ciphers to encrypt/decrypt inputs, using keys and transformations on the input (bitwise operations, permutations etc…)

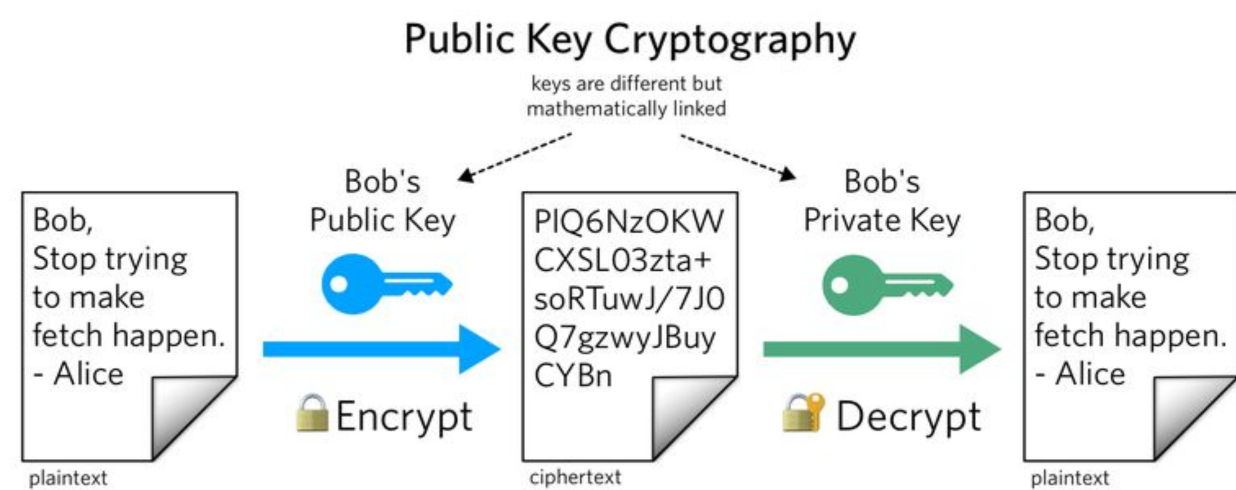

Figure 1. Cryptography for encrypting/decrypting a file

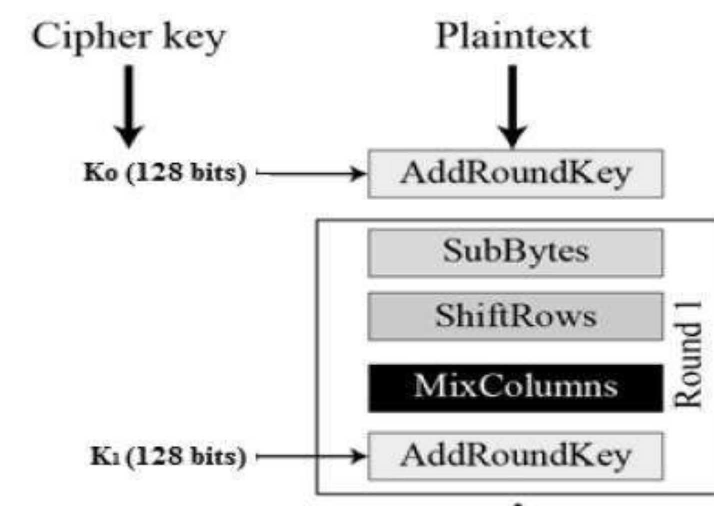

Figure 2. A round of transformation for encryption for AES (Advanced Encryption Standards)

Collecting the dataset

The problem was narrowed down to file-level binary classification for C/C++.

V1: files from crypto competition submissions, crypto libraries (positive examples) and files from algorithmic competitions and randomly selected github repositories (negative examples). V1 has a few important shortcomings, including:

- easily separable classes,
- lack of diversity and coverage

V2: contains OS code, hashing, signal processing, networks, bitwise based operations, ML and heavily math-based non-crypto examples - with an emphasis on keeping ambiguous files in the mix instead of discarding them.

```c
void
_nettle_aes_encrypt(unsigned rounds, const uint32_t *keys,
                    const struct aes_table *T,
                    size_t length, uint8_t *dst,
                    const uint8_t *src)
{
  FOR_BLOCKS(length, dst, src, AES_BLOCK_SIZE)
    {
      uint32_t w0, w1, w2, w3;          /* working ciphertext */
      uint32_t t0, t1, t2, t3;
      unsigned i;

      /* Get clear text, using little-endian byte order.
       * Also XOR with the first subkey. */

      w0 = LE_READ_UINT32(src)      ^ keys[0];
      w1 = LE_READ_UINT32(src + 4)  ^ keys[1];
      w2 = LE_READ_UINT32(src + 8)  ^ keys[2];
      w3 = LE_READ_UINT32(src + 12) ^ keys[3];

      for (i = 1; i < rounds; i++)
        {
          t0 = AES_ROUND(T, w0, w1, w2, w3, keys[4*i]);
          t1 = AES_ROUND(T, w1, w2, w3, w0, keys[4*i + 1]);
          t2 = AES_ROUND(T, w2, w3, w0, w1, keys[4*i + 2]);
          t3 = AES_ROUND(T, w3, w0, w1, w2, keys[4*i + 3]);

          /* We could unroll the loop twice, to avoid these
             assignments. If all eight variables fit in registers,
             that should give a slight speedup. */
          w0 = t0;
          w1 = t1;
          w2 = t2;
          w3 = t3;
        }

      /* Final round */

      t0 = AES_FINAL_ROUND(T, w0, w1, w2, w3, keys[4*i]);
      t1 = AES_FINAL_ROUND(T, w1, w2, w3, w0, keys[4*i + 1]);
      t2 = AES_FINAL_ROUND(T, w2, w3, w0, w1, keys[4*i + 2]);
      t3 = AES_FINAL_ROUND(T, w3, w0, w1, w2, keys[4*i + 3]);

      LE_WRITE_UINT32(dst, t0);
      LE_WRITE_UINT32(dst + 4, t1);
      LE_WRITE_UINT32(dst + 8, t2);
      LE_WRITE_UINT32(dst + 12, t3);
    }
}
```

Crypto example

```c
void Sha512_data(SHA512_DATA *sha, const void *buffer, DWORD len)
{
	DWORD templen;

	/* Add to the total length of the input stream */
	sha->totalLen += (QWORD)len;

	/* Copy the blocks into the input buffer and process them */
	while(len > 0)
	{
		if(!(sha->inputLen) && len >= 128)
		{
			/* Short cut: no point copying the data twice */
			ProcessBlock(sha, (const BYTE *)buffer);
			buffer = (const void *)(((const BYTE *)buffer) + 128);
			len -= 128;
		}
		else
		{
			templen = len;
			if(templen > (128 - sha->inputLen))
			{
				templen = 128 - sha->inputLen;
			}
			memcpy(sha->input + sha->inputLen, buffer, templen);
			if((sha->inputLen += templen) >= 128)
			{
				ProcessBlock(sha, sha->input);
				sha->inputLen = 0;
			}
			buffer = (const void *)(((const BYTE *)buffer) + templen);
			len -= templen;
		}
	}
}
```

Non-crypto example (non-trivial)

Figure 3. Sample data

Models

Two different approaches to code processing - how to capture signal from code?:

- Model A - using hand-crafted features and metadata features with various counts of code elements (loops, bitwise operations, type declarations), imports…
- Model B - Processing the code as a text input by using an embedding to turn the code into a fixed length vector

Benchmark: pattern matching on code as text (using generic terms like "crypt" as well as calls to crypto libraries and mentions of registered crypto algorithms and protocols) - using Wind-River's crypto-detector[1]

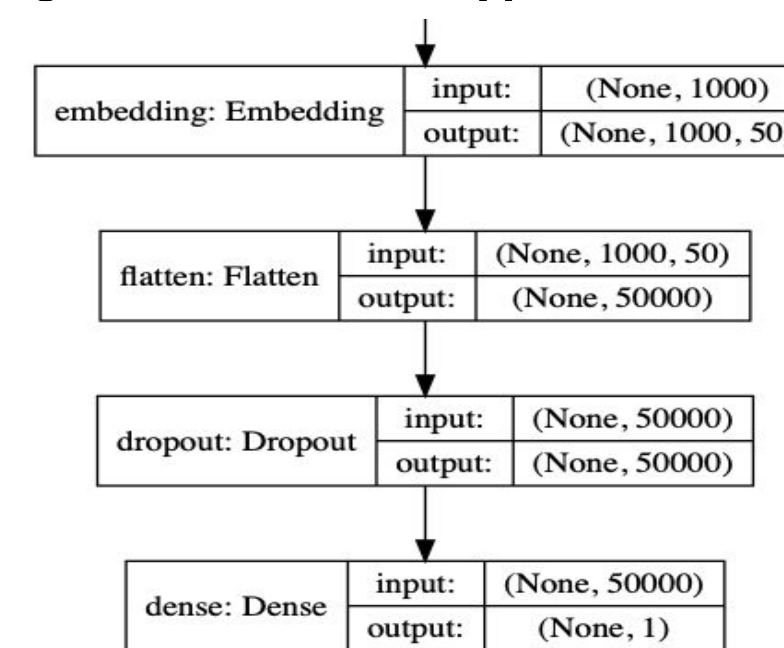

Figure 4. Architecture of the embedding-based model

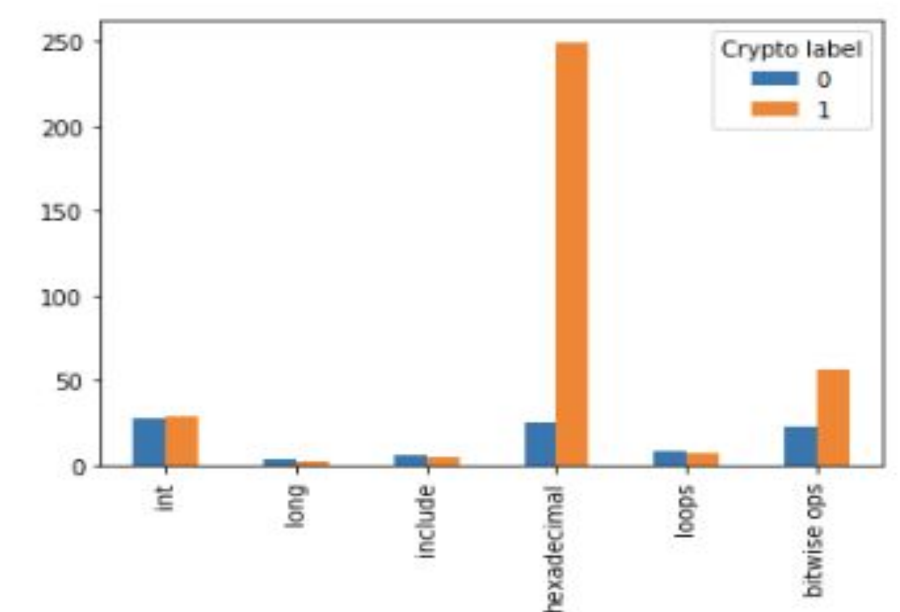

Figure 5. Counts of different code features across the two classes in V1

Results

Classification results from our models:

Model	Precision	Recall	F1	F2
Model A	0.75	0.89	0.81	0.86
Model B	0.90	0.87	0.88	0.88
Benchmark	0.79	0.87	0.83	0.86

The benchmark already performs well on our dataset, and Model B only improves slightly on it.

Limitations:

- Dataset: the model performance is bound by the dataset quality
- Hard cases: hashing functions that are very similar to cryptography code

Potential improvements:

- Collect files from more varied sources
- Build syntactic trees to capture the semantics of the code before embedding

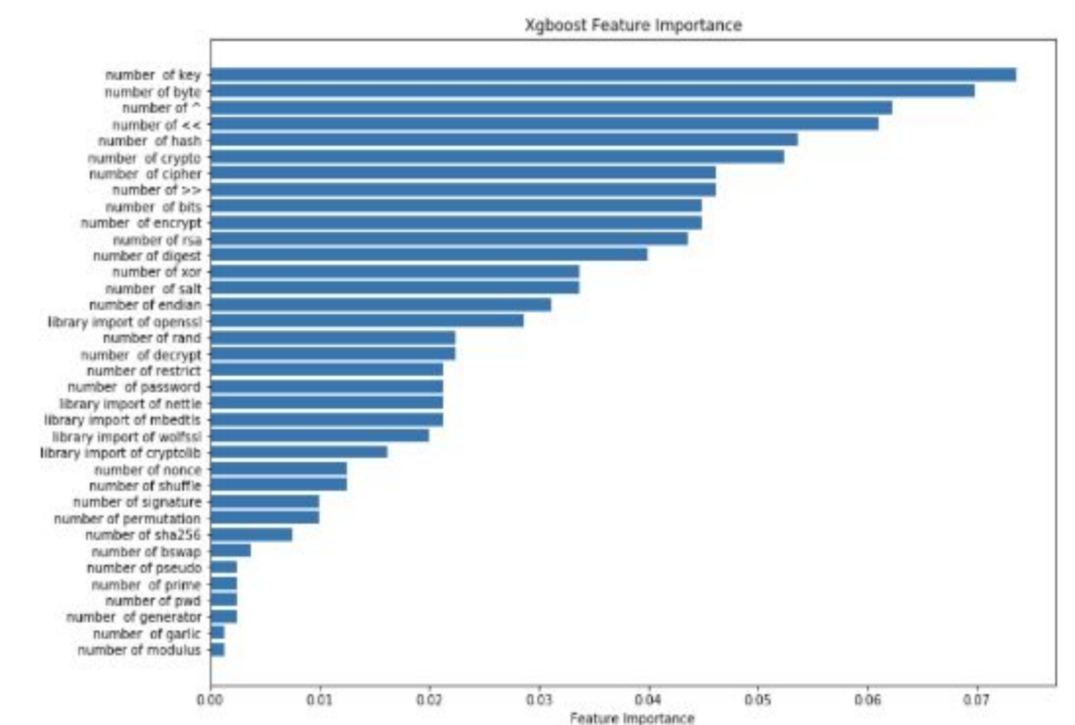

Figure 6. Feature importance of hand-crafted features using boosted trees

Reference

[1]Wind-river: cryptography detection tool: https://github.com/Wind-River/crypto-detector