Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK

Dinh Duy Kha

Overview

- The paper define crypto buffer and crypto operations, then use Static Analysis (Source Code Analysis) to automatically determine crypto buffers and crypto operations from the initial list of operations. This is somewhat simlilar to

several works that use Taint Analysis to propagate secret variables (PtrSplit,DataShield: Configurable Data Confidentiality and Integrity)

- The crypto domain is protected with Intel MPK. Switches between the domains is automatically inserted. Stack and heap allocations is replaced with a customized implementation

Crypto buffer and crypto operations

- **Crypto buffer:** memory regions that contains the secret key, or the intermediate results of cryptographic computations
- **Crypto operations:** Operations/instructions that can access the crypto buffers. For example, a memory read instruction to the AES key.

Crypto operations

The paper points out that crypto operations have interesting properties.

1. Modern cryptographic operations, on top of the secret keys, use a lot of **intermediate values** such as buffers and runtime data. Those data also need to be protected, but often overlooked by most mechanisms.
2. Cryptographic operation have data flow that declassify the information. For instance, ciphertext have data flow from the key, but is considered non-confidential. Traditional Taint analysis (DataShield: Configurable Data Confidentiality and Integrity) would over-taint them (making ciphertext also tainted), thus:

a. Have more performance overhead
b. Increase the TCB

Process

- **Pre-analysis**: Compile the program that contain the annotation into LLVM IR
- **Crypto buffer labeling**: Use Points-to Analysis and Taint Analysis to label crypto buffers.
- **Crypto operations identification**: Identify memory accesses and operations that access the crypto buffers
- **Code transformation**:
  1. Move crypto buffers into protected memory regions
  2. Replicate functions to be protected (why?)
  3. Partition program into crypto domain and non-crypto domain
  4. Instrument switches when crossing boundary
  5. Additional binary security checks

Crypto buffer labeling

The labeling of crypto buffers is similar to that of Taint Analysis . The developers have to annotate which values (in this case only the pointers?) are sensitive (the encryption key or private key). CryptoMPK uses the initial annotations as the taint source and propagates the taint through data flow. The taint here is called the "crypto" tag.

Points-to Analysis

Points-to Analysis (Pointer Analysis) is used to "improve the accuracy of both control-flow and data-flow construction" (Not sure the implication here).

Crypto-aware

The paper makes the taint analysis "crypto-aware" by allowing the developer to additionally annotate plaintext and ciphertext buffers with the "multulally-exclusive (mxor)" tag. During taint analysis, when data flow to those buffer, the crypto tag will be eliminated.

Context-sensitive analysis

It also uses Context-sensitive Analysis to have different tags for the same buffer, under different contexts. The reasoning is that some function have different sensitivity, based on the calling context.

Crypto operations identification

This step identifies which part of the program are allowed to access the crypto buffers. Those operations are the LLVM IR load and store instructions, and the memory allocation and deallocation operations (malloc).

Memory access: When the load and store instruction access the crypto buffer's address, it is considered a crypto operations

Memory management: When the memory allocation and deallocation target the crypto buffer, it is considered a crypto operation. Then, it create a custom malloc/free for the crypto context

This is kind of similar to Cali: Compiler-Assisted Library Isolation, where the allocation is replaced to use shared memory when needed. However, here, every different context have different allocation function (context-sensitive)(7.2).

Code transformation

Finally, after having the crypto buffers and the crypto operations, it transforms the source code at LLVM IR level.

Memory allocations

Stack and heap allocation of crypto bufers is replaced to allocate inside protected memory (Protected by Intel MPK).

Ambiguous functions that contains memory allocation ATTACH

```
1   char *xreadline(int fd) {
2     char *buf = malloc(MAX_LEN);
3     for (int i = 0; i < MAX_LEN; i++) {
4       int n = read(fd, buf+i, 1);
5       if (n <= 0 || buf[i] == '\n') return buf;
6     }
7     return buf;
8   }
9   ---------------------------
10  char *readkey() {
11    [...]
12    char *line = xreadline(fd);
13    [...]
14    return line;
15  }                          → Sensitive context
16
17  char *prompt() {
18    [...]
19    char *line = xreadline(fd);
20    [...]
21    return line;
22  }                          → Non-sensitive context
```

Listing 2: An example of context-dependent memory allocation

For functions that (1) contains memory allocations and (2), could be either sensitive or non-sensitive, depending on the calling context, there must be multiple versions of the function for each context. This is because inside those functions, the permissions for each instructions could be different on each calling context (the parameters for the functions). The paper duplicate all of such functions, each with different security permissions, and insert them to the original calling context.

Moreover, to reduce the number of duplicated functions, the paper compare the signature of the calling context, and merge duplicated functions with identical signature.

Context switches

The paper use different protection granularity for the Domain Switching in Intra-process Isolation. Two granularities are used, instruction level and function level. Function level is used for **hotspot functions**, which are functions that match the proposed heuristic. The heuristic calculate a score that is roughly based on the number of sensitive instructions over the number of total instructions, with higher weight on sensitive instructions inside of loops and calling instructions.

There is no parameter passing from one domain to another, because the paper only cares about Load and Store instructions and memory allocation functions (see 6). That is, all sensitive memory allocation/deallocation must be replaced, and all sensitive Load and Store (to crypto buffer) must be inside the sensitive domain.

Notes

- Design and implementation fragment the paper and make it hard to understand.
- Only load and store to the identified crypto buffers are protected.
- Only target crypto libraries.