August 2020

Security by Obfuscating Data Across Non-Adjacent Memory Locations

Aaron Vaage

Security by Obfuscating Data Across Non-Adjacent Memory Locations

ABSTRACT

The bytes that make up a data primitive (e.g., float, integer, etc.) are stored in adjacent bytes and in known order, based on a computer's architecture. Storing data in adjacent bytes is required by the hardware in order to operate on them. Compilers, in an effort to improve cache efficiency, pack primitives together. This packing indirectly communicates the relationship between primitives. When reverse engineering code, an attacker can observe the state of data on the stack. Knowing how the data is structured makes this much easier. The most reliable structuring is knowing how primitives will be structured. This disclosure describes techniques that achieve code security by distributing data across distinct, non-adjacent, randomly-selected memory locations, thereby reducing the accessibility of the data to attackers.

KEYWORDS

- Computer security
- Code security
- Cybersecurity
- Reverse engineering
- Data obfuscation
- Memory localization
- Non-adjacent memory

BACKGROUND

The bytes that make up a data primitive (e.g. float, integer, etc.) are stored in adjacent bytes and in known order, based on a computer's architecture. Storing data in adjacent bytes is required by the hardware in order to operate on them. Compilers, in an effort to improve cache efficiency, pack primitives together. This packing indirectly communicates the relationship between primitives. When reverse engineering code, an attacker will observe the state of data on the stack. Knowing how the data is structured makes this much easier. The most reliable structuring is knowing how primitives will be structured.

While traditional code obfuscation can make it harder for an attacker to understand the code's behavior, high-level behaviors can be determined by monitoring changes in memory. A common technique for data obfuscation is using mixed Boolean-arithmetic expressions to break basic expressions into more complex expressions. White-box AES combines and encodes sensitive information and operations into tables such that sensitive information cannot be extracted. While these existing techniques provide an effective means of hiding sensitive information, they retain the standard primitives structures.

DESCRIPTION

This disclosure describes techniques that achieve code security by distributing data across distinct, non-adjacent, and randomly-selected memory locations, thereby reducing the accessibility of the data to attackers. The effective application of these techniques requires a trade-off of performance in favor of security.

```
template <size_t width>
struct View {
uint8_t* bytes[width];
};
```

(a)

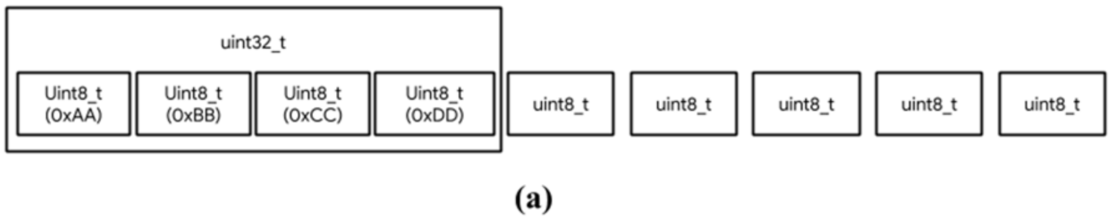
```
uin8_t buffer[128];

View<4> x;
x.bytes[0] = buffer + 32;
x.bytes[1] = buffer + 3;
x.bytes[2] = buffer + 27;
x.bytes[3] = buffer + 73;

View<4> y;
y.bytes[0] = buffer + 79;
y.bytes[1] = buffer + 59;
y.bytes[2] = buffer + 104;
y.bytes[3] = buffer + 4;
```

(b)

Fig. 1: (a) A view; (b) Two views, `x` and `y`, that reference different parts of the buffer without trampling on each other

Per the techniques, illustrated by the pseudo-code of Fig. 1(a), a view is defined as a group of non-adjacent memory locations that should be considered a single piece of contiguous memory. A view is an index to reassemble to individual bytes of a data primitive. Using a reserved section of memory (`buffer`, of Fig. 1(b)), multiple views, e.g., `x` and `y`, can reference different bytes of the reserved section as part of themselves.

For example, `x` and `y` represent data primitives, e.g., of type unsigned 32-bit integers (`uint32`), that each occupy four bytes. Constituent bytes of `x` (or `y`) occupy distinct, randomly-selected, non-adjacent bytes of the reserved memory section. In the example of Fig. 1(b), the constituent bytes of `x` occupy offsets `32`, `3`, `27`, and `73` of the reserved memory section. To avoid different views from trampling on each other, each byte in the reserved memory section may only be referenced (at most) once. Thus the constituent byes of `y` occupy offsets `79`, `59`, `104`, and `4`, different from the offsets occupied by `x`.



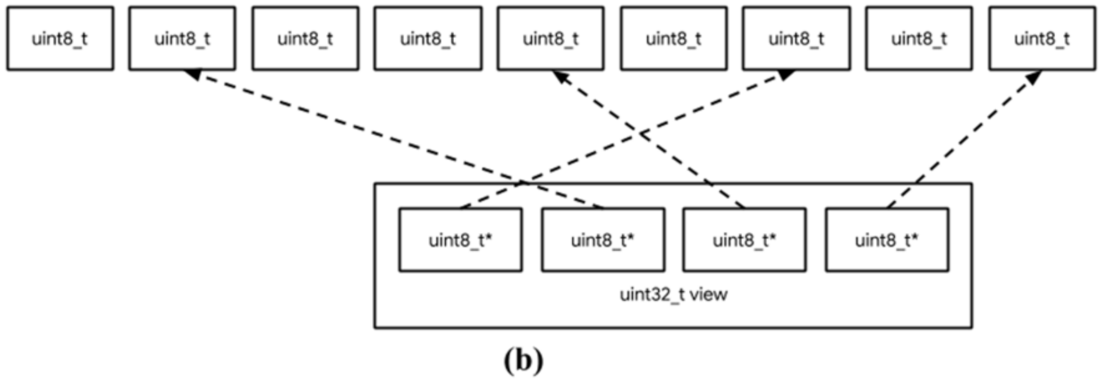


Fig. 2: (a) An unobfuscated data primitive (b) A view, e.g., an obfuscated data primitive

Fig. 2 illustrates a graphical view of an unobfuscated data primitive and its corresponding view, e.g., obfuscated data primitive. Fig. 2(a) illustrates a data primitive that is allocated and stored in contiguous memory, e.g., adjacent memory locations, as is done conventionally. If an attacker knows what some of the data should look like, by looking at a memory dump they can find and extract sensitive data.

Fig. 2(b) illustrates a view, e.g., an obfuscated data primitive. Per the techniques, the view stores the addresses to the constituent bytes of the data primitive, the bytes themselves being stored in distinct, randomly-selected, non-adjacent memory locations. By grouping together addresses of non-adjacent bytes, they can be referenced in order without being stored in order.

Operations on views

Since the data is no longer stored in adjacent memory, standard operations (e.g. additions) would require copying the data into the standard, architecture-defined structure, revealing the data in memory. Instead, it is possible to define an operation that operates on the view that avoids revealing the value in memory, as follows.

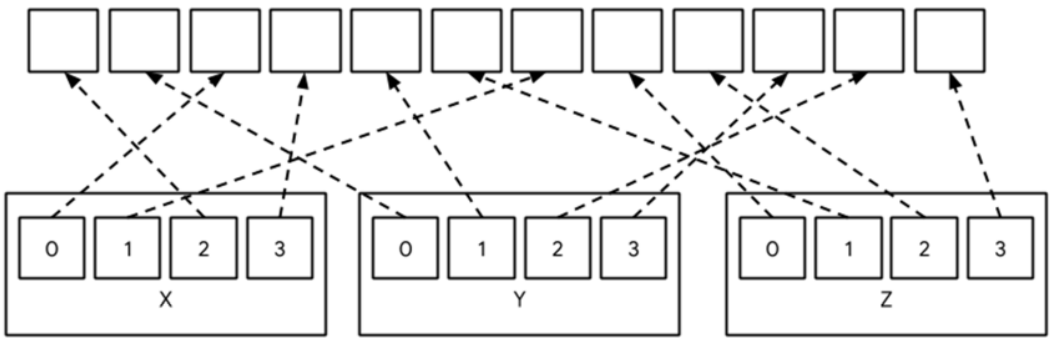
```
uint16_t temp = 0;
temp += x.bytes[0];

temp += y.bytes[0];
out.bytes[0] = (uint8_t)(0xFF & temp);

temp = temp >> 8;
temp += x.bytes[1];
temp += y.bytes[1];
out.bytes[1] = (uint8_t)(0xFF & temp);

temp = temp >> 8;
temp += x.bytes[2];
temp += y.bytes[2];
```

```
out.bytes[2] = (uint8_t)(0xFF & temp);

temp = temp >> 8;
temp += x.bytes[3];
temp += y.bytes[3];
out.bytes[3] = (uint8_t)(0xFF & temp);
```

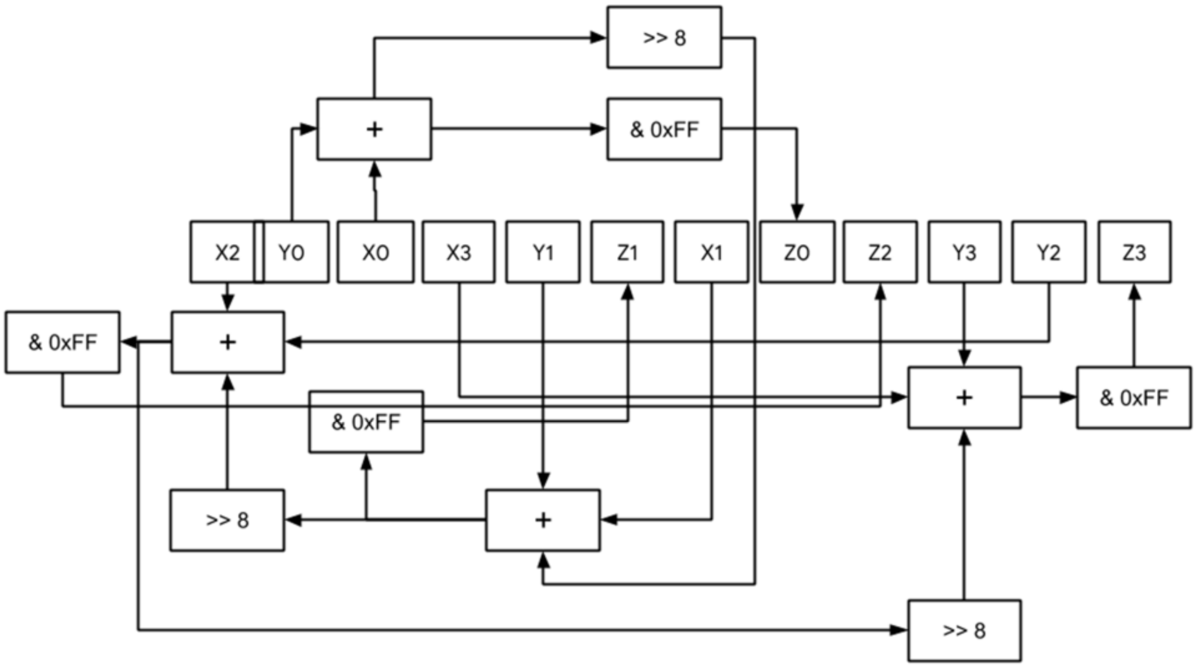
Fig. 3: Illustrating operations on views, e.g., the addition of two `uint32` views `x` and `y`

For example, Fig. 3 illustrates the operation of addition on two `uint32` variables `x` and `y` that have been obfuscated into views. The first bytes of `x` and `y` are dereferenced (`x.bytes[0]` and `y.bytes[0]`), and their sum stored in a temporary variable (`temp`). The least significant byte of the sum is extracted (by masking with `0xFF`) and stored in the first byte of an output variable `out`, which is itself a view, e.g., an index to reassemble an output variable of type `uint32`.

The most significant byte of the sum, which acts as overflow towards the second byte-addition, is extracted by right-shifting `temp` by eight bits. The second, third, and fourth bytes of `x` and `y` are similarly dereferenced, added (along with the overflow from the previous byte-addition), masked, and stored in the appropriate byte of the output view. The operations of Fig. 3 are executed without moving the operands `x` and `y` into adjacent memory locations, further reducing the possibility of data exposure. The multiple operations of Fig. 3, which supplant a relatively simple operation such as addition, make it difficult for an attacker to discern a pattern in data movements within memory.



(a)



(b)

Fig. 4: Illustrating an addition operation on two uint32 views, `x` and `y`, to produce a third uint32 view `z` (a) The views `x`, `y`, and `z` (b) Operations on the constituent bytes of `x` and `y` to produce `z`

Fig. 4 illustrates graphically an addition operation on two `uint32` views, `x` and `y`, to produce a third `uint32` view, `z`. Fig. 4(a) illustrates the views `x`, `y`, and `z`, showing the mapping of their constituent bytes to randomly-selected, non-adjacent memory locations within a buffer. Fig. 4(b) illustrates the operations on the views `x` and `y` to add them and store them in view `z`.

As explained earlier, the constituent bytes of `x` and `y` are dereferenced, added (along with the overflow from the previous byte-addition), masked, and stored in the appropriate location of view `z`. The operations of Fig. 4(b) can be executed without moving the operands `x` and `y` into adjacent memory locations, further reducing the possibility of data exposure. The multiple operations represented by Fig. 4(b), which supplant a relatively simple addition operation, make it difficult for an attacker to discern a pattern in data movements within memory.

The operations on views, e.g., the addition operation illustrated here, can be designed to achieve specific trade-offs between computational speed and data obfuscation. Data obfuscation, as described herein, can be done at compile time. For example, a compiler can automatically translate primitives into predefined views, such that primitives within some scope are mapped into distinct, non-adjacent memory locations randomly selected by the compiler.

CONCLUSION

This disclosure describes techniques that achieve code security by distributing data across distinct, non-adjacent, randomly-selected memory locations, thereby reducing the accessibility of the data to attackers.

REFERENCES

[1] Eyrolles, Ninon. “Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis, and Simplification tools.” PhD dissertation, Université Paris-Saclay, 2017.