`unsafe` Cryptography: Common Vulnerabilities in Modern Programming Languages

ICMC 2024

Paul Bottinelli [paul.bottinelli@nccgroup.com] - Cryptography Services

nccgroup

Cryptography reviews: where do we find vulns?

Randomness

Algorithm Confusion

Key Management

Crypto Primitives and Protocols

Side-channel Attacks

Programming

Cryptography reviews: where do we find vulns?

Randomness

Algorithm Confusion

Key Management

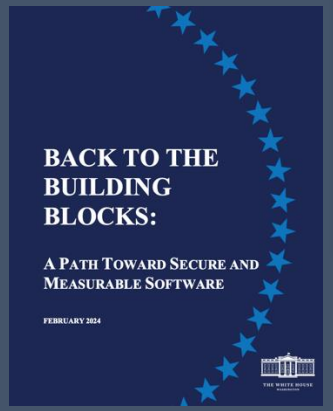
Crypto Primitives and Protocols

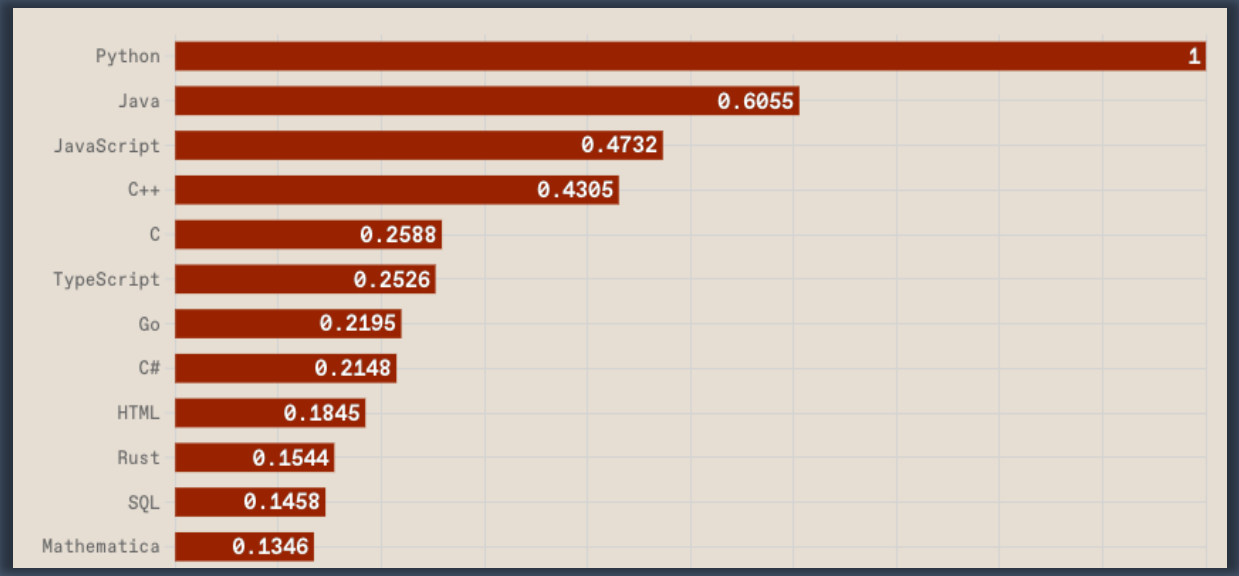
Side-channel Attacks

Programming

Common Vulnerabilities in Modern Programming Languages

- Modern Languages
  - Memory Safe, Extensive Standard Library, etc
- White House cybersecurity report calling for memory-safe languages to replace C and and C++
- Rust called out in report

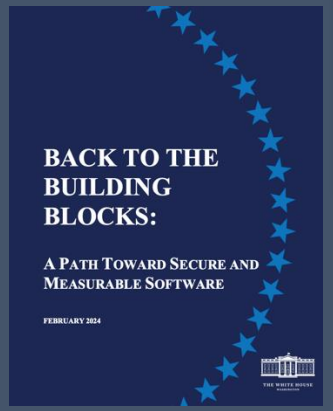


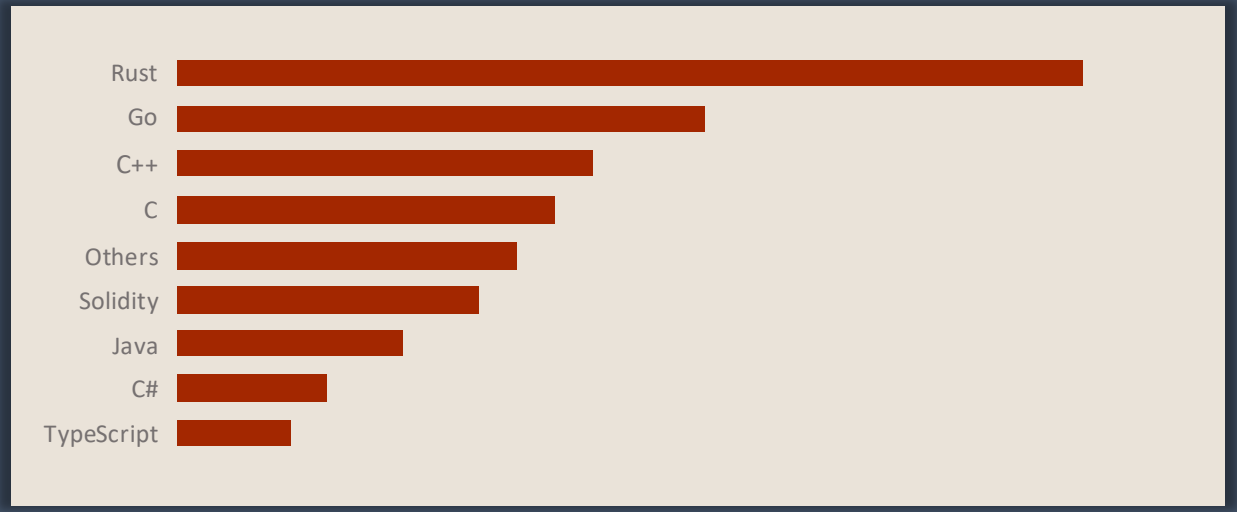


The Top Programming Languages 2024: *Typescript and Rust are among the rising stars* (2024, Aug 22). https://spectrum.ieee.org/top-programming-languages-2024

Common Vulnerabilities in Modern Programming Languages

- Modern Languages
  - Memory Safe, Extensive Standard Library, etc
- White House cybersecurity report calling for memory-safe languages to replace C and and C++
- Rust called out in report

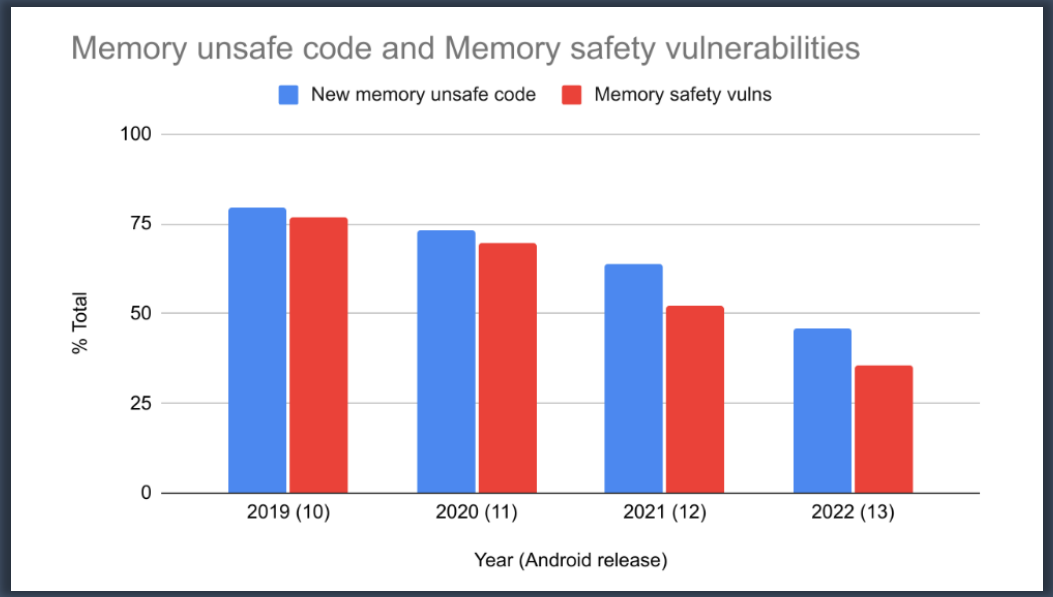




Approximate, localized, anecdotal and contextual language distribution from NCC Group's Cryptography Services reviews

Case study - Memory Safe Languages in Android 13

- Rust supported since Android 12
- Drop in memory safety vulns:
  - 2019: 223 -> 2022: 85
- “Android 13 is the first Android release where a majority of new code added to the release is in a memory safe language”

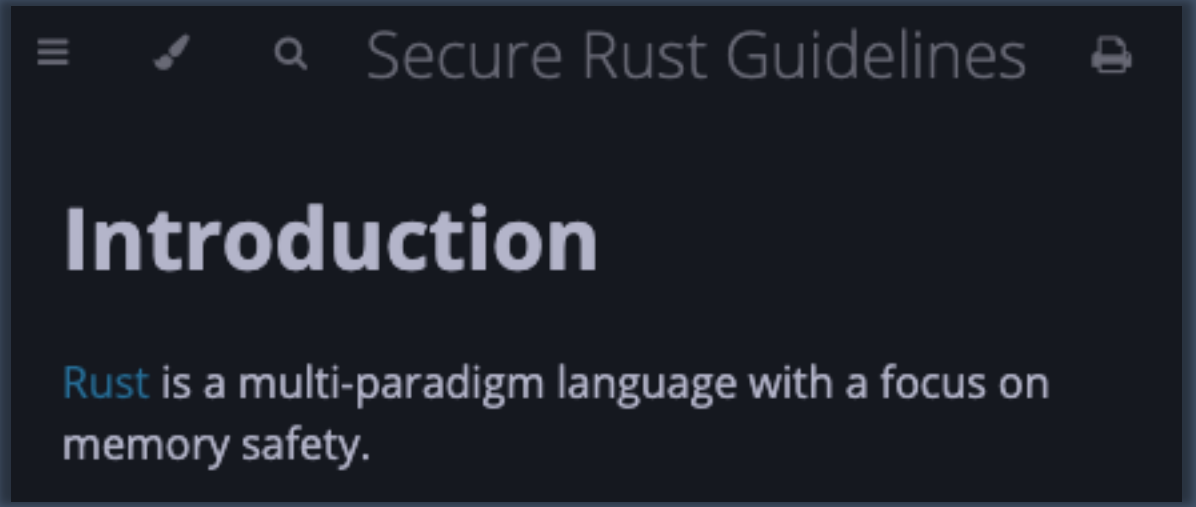


https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

- “2022 is the first year where memory safety vulnerabilities do not represent a majority of Android’s vulnerabilities”

Rust and Secure Coding

- This talk uses Rust as illustration, parallels can be drawn to other languages
- Secure Rust Guidelines[1]
  - `Cargo-outdated` and `Cargo-audit`
  - Integer overflows
  - Panics
  - `unsafe` code
  - etc
- Nothing about specific bugs in that resource



[1] https://anssi-fr.github.io/rust-guide/

1. Generation of random vectors

Rust `Vec` – a contiguous growable array type, written as `Vec<T>`, short for 'vector'.

```
let mut vec = Vec::new();
vec.push(1);
vec.push(2);                // vec = [1, 2]

// The `vec!` macro is provided for convenient initialization:
vec = vec![1, 2, 3];        // vec = [1, 2, 3]

// It can also initialize each element of a `Vec<T>` with a given value.
// This may be more efficient than performing allocation and
// initialization in separate steps, especially when initializing
// a vector of zeros:
vec = vec![0; 5];           // vec = [0, 0, 0, 0, 0]
```

1. Generation of random vectors

- Code snippet *attempts to* generates a vector of 8 random elements

```
let mut rng = rand::thread_rng();
let v: Vec<u8> = vec![rng.gen(); 8];

println!("{:?}", v);
```

1. Generation of random vectors

- Code snippet *attempts to* generates a vector of 8 random elements

```
let mut rng = rand::thread_rng();
let v: Vec<u8> = vec![rng.gen(); 8];

println!("{:?}", v);
// > [42, 42, 42, 42, 42, 42, 42, 42]
```

- *Only one* random element is sampled and copied 8 times
- Fix:

```
let v: Vec<u8> = (0..8).map(|_| rng.gen()).collect();
```

2. Overzealous functional programming constructions

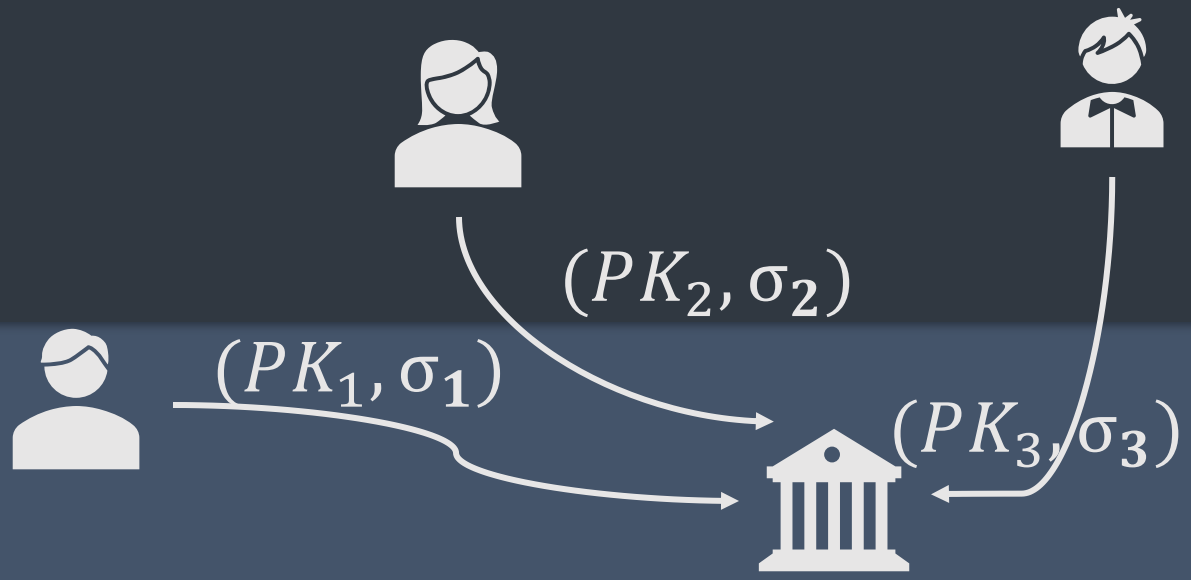
- Complex usage of `zip()`, `map()`, `filter()`, etc.
- Example of `zip()`:

```
['a', 'b', 'c'].zip([1, 2, 3]) => [('a', 1), ('b', 2), ('c', 3)]
```

- These operators have intricacies:
  - E.g. zipping unbalanced lists returns early
- Classic example: signature bypass in multi-signature schemes

2. Overzealous functional programming constructions

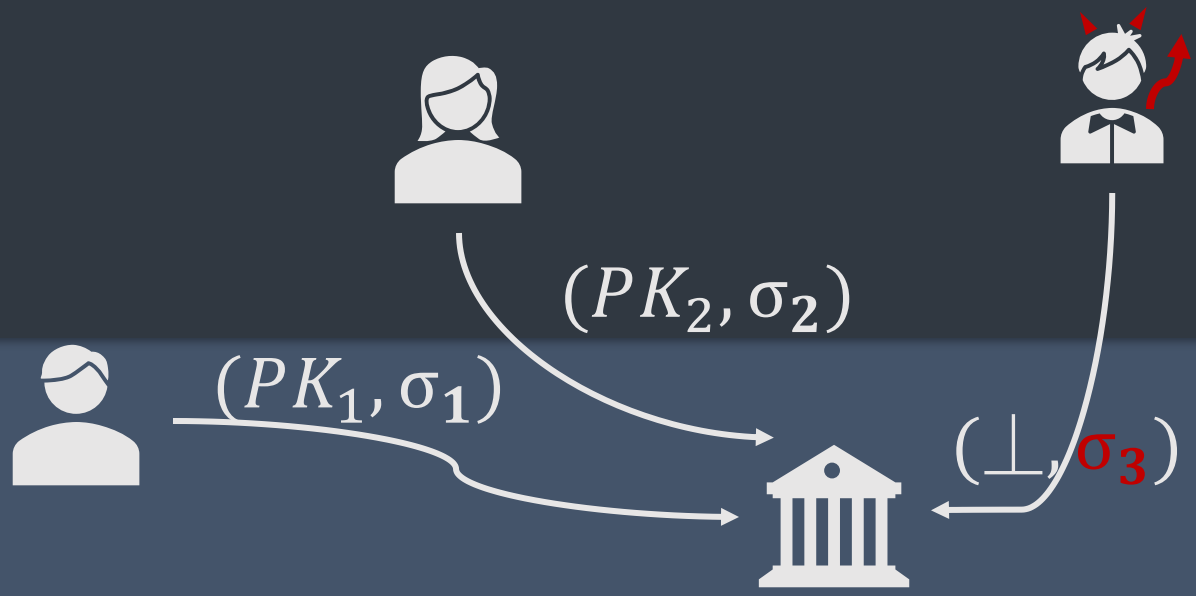
```
fn verify(public_keys: Vec<PublicKey>, signatures: Vec<Signature>, msg: Hash) -> bool {
    for key, sig in public_keys.zip(signatures) {
        if !key.verify(sig, msg) {
            return false;
        }
    }
    return true;
}
```



n-out-of-n verifier

2. Overzealous functional programming constructions

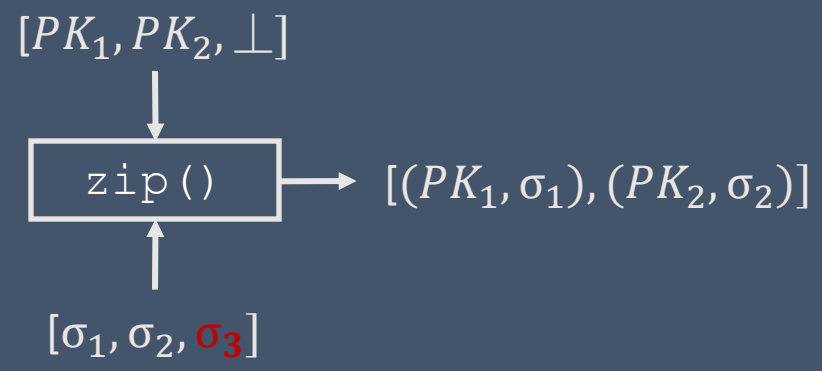
```
fn verify(public_keys: Vec<PublicKey>, signatures: Vec<Signature>, msg: Hash) -> bool {
    for key, sig in public_keys.zip(signatures) {
        if !key.verify(sig, msg) {
            return false;
        }
    }
    return true;
}
```

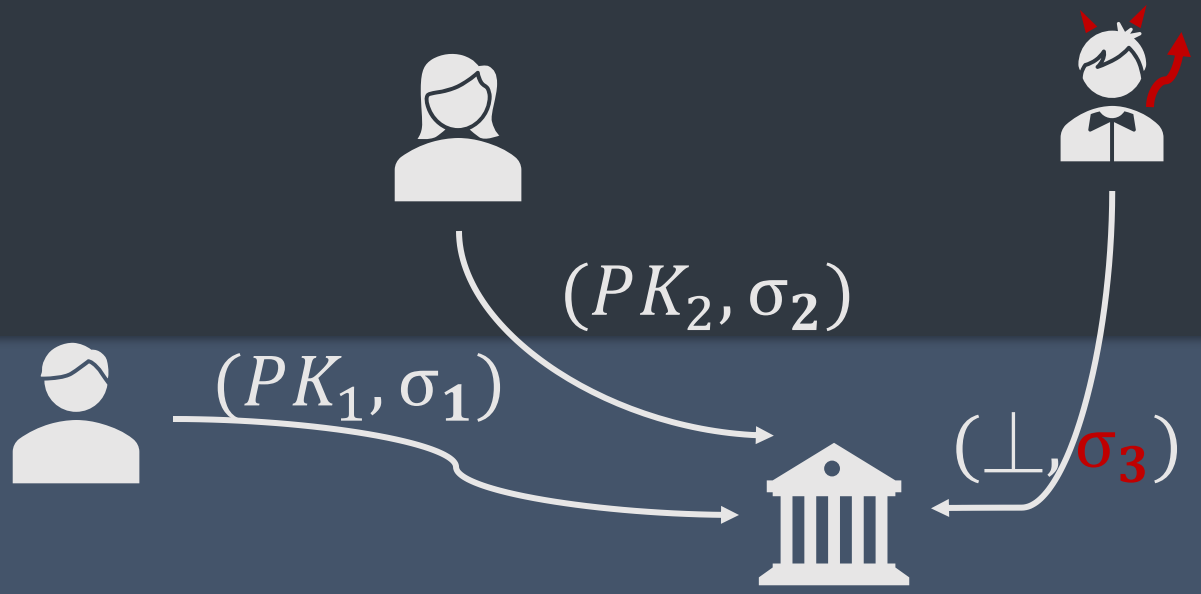


n-out-of-n verifier

2. Overzealous functional programming constructions

```
fn verify(public_keys: Vec<PublicKey>, signatures: Vec<Signature>, msg: Hash) -> bool {
    for key, sig in public_keys.zip(signatures) {
        if !key.verify(sig, msg) {
            return false;
        }
    }
    return true;
}
```

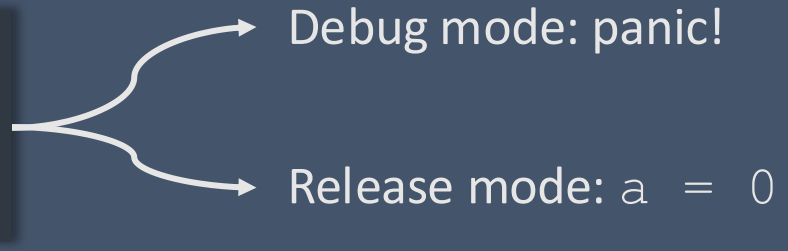




3. Release vs Debug Mode

- Using debug or release compilation profile changes integer overflow behavior
  - In debug configuration, overflow cause the termination of the program (panic)
  - In release configuration, the computed value silently wraps around

```
let mut a: u8 = 255;
a += 1;
```



- *Instead*, use integer arithmetic operations that have the same behavior in debug and release mode, such as
  `saturating_add(a, b)`: computes a + b, saturating at numeric bounds

4. Inconsistencies between languages

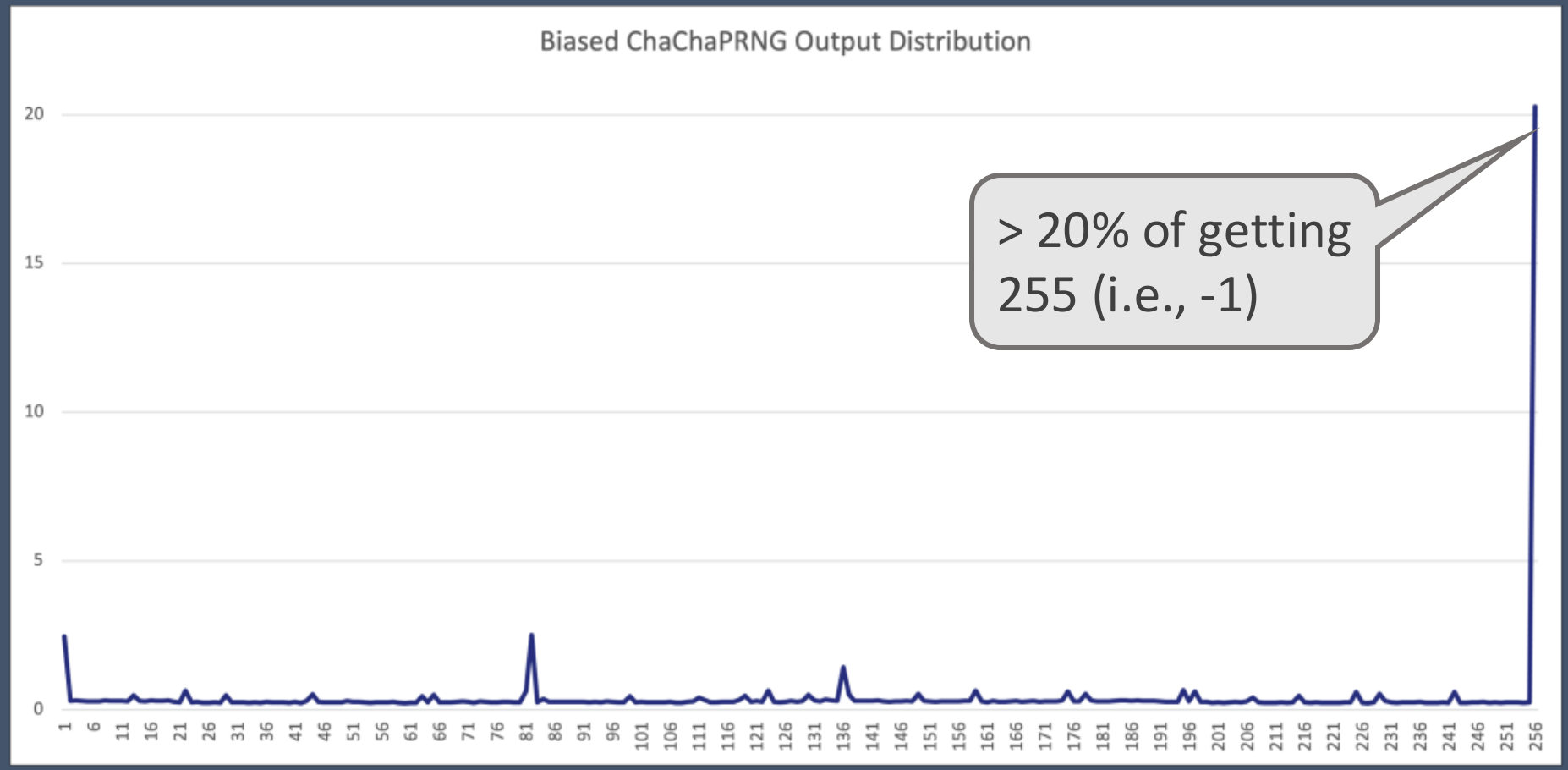
- Rust ! when applied to integers is different than C
- In Rust, ! on integers performs binary negation (equivalent to ~ in C)
- In C, ! on integers enforces Boolean interpretation
  - Rust would not allow that because of strong typing
- May be hard to catch because of macro syntax which also uses !
- Example: signature verification bypass

```
ensure!(!signatures.len() >= 2, "Bundle must
    have at least 2 signatures");
```

4. Inconsistencies between languages

- Rust has native support for unsigned integer types (e.g., `u8`, `u32`, etc.)
  - Arithmetic right shift on signed integer types, logical right shift on unsigned integer types
- Java does not have primitive types for unsigned integers, values are *interpreted in two's complement*
  - the ">>" operator performs a *signed right shift.* The sign bit is maintained and not unconditionally replaced by a zero
  - 0b1000 >>> 1 = 0b0100 (unsigned right shift)
  - 0b1000 >> 1 = 0b1100 (signed right shift)
- PRNG implementation translated to Java made this mistake

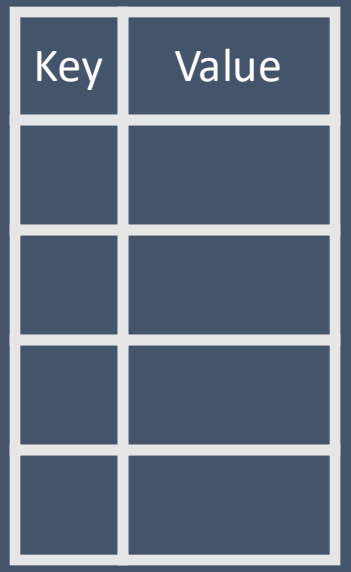
4. Inconsistencies between languages



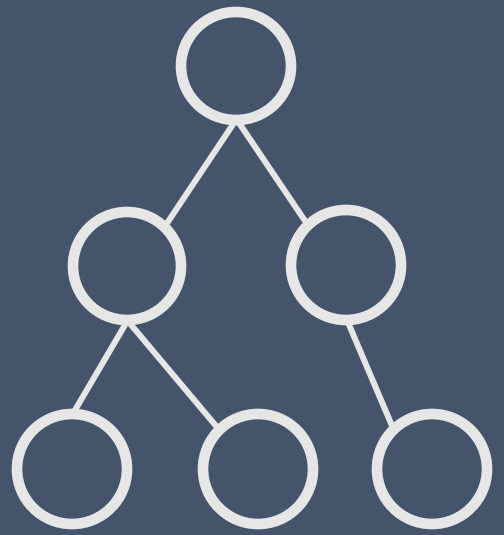
Dancing Offbit: The Story of a Single Character Typo That Broke a ChaCha-based PRNG

5. Map, Tree and Set data structures

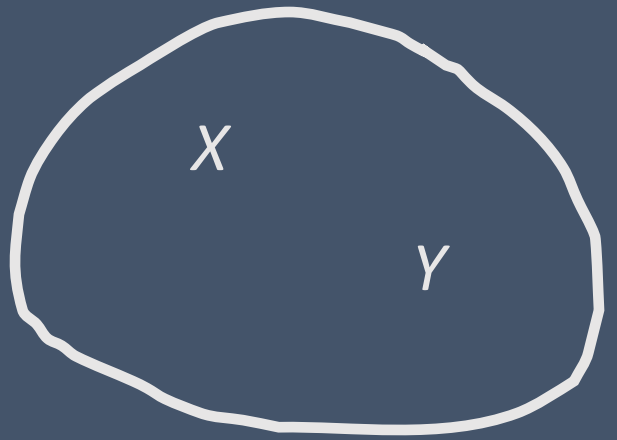
Rust provides several advanced data structures (https://doc.rust-lang.org/std/collections/)



Map

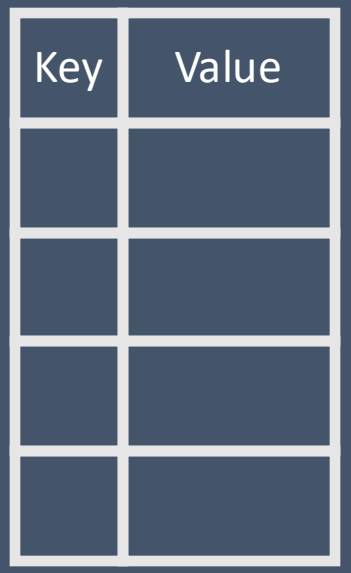


Tree



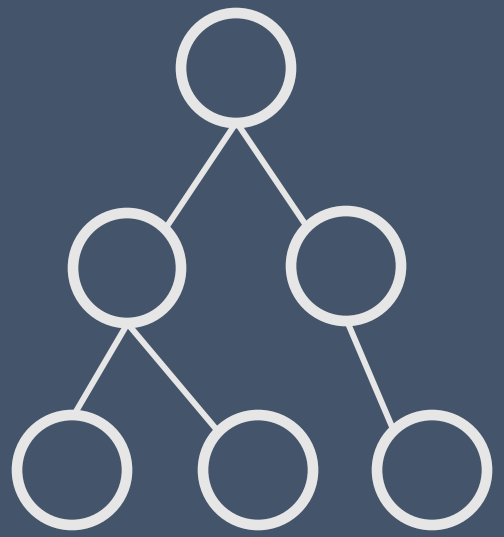
Set

5. Map, Tree and Set data structures



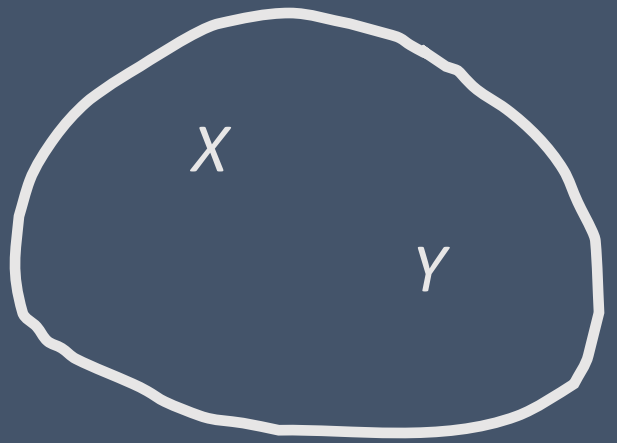
Map

`HashMap, IndexMap`



Tree

`BTreeMap, BTreeSet`



Set

`HashSet, IndexSet`

5. Set, Map and Tree data structures

`HashMap`: hash table where the iteration order depends on the hash values of the keys

```
pub fn iter(&self) -> Iter<'_, K, V>      1.0.0 · source
```

ⓘ

An iterator visiting all key-value pairs in arbitrary order. The iterator element type is `(&'a K, &'a V)`.

`IndexMap`: hash table where the iteration order of the key-value pairs is independent of the hash values of the keys

The key-value pairs have a consistent order that is determined by the sequence of insertion and removal calls on the map. The order does not depend on the keys or the hash function at all.

All iterators traverse the map in *the order*.

The insertion order is preserved, with **notable exceptions** like the `.remove()` or `.swap_remove()` methods. Methods such as `.sort_by()` of course result in a new order, depending on the sorting order.

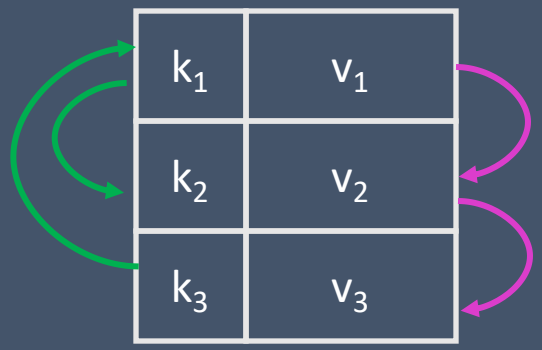
5. Set, Map and Tree data structures

`HashMap` : hash table where the iteration order depends on the hash values of the keys

```
pub fn iter(&self) -> Iter<'_, K, V>      1.0.0 · source
```

ⓘ

An iterator visiting all key-value pairs in arbitrary order. The iterator element type is `(&'a K, &'a V)`.



`IndexMap` : hash table where the iteration order of the key-value pairs is independent of the hash values of the keys

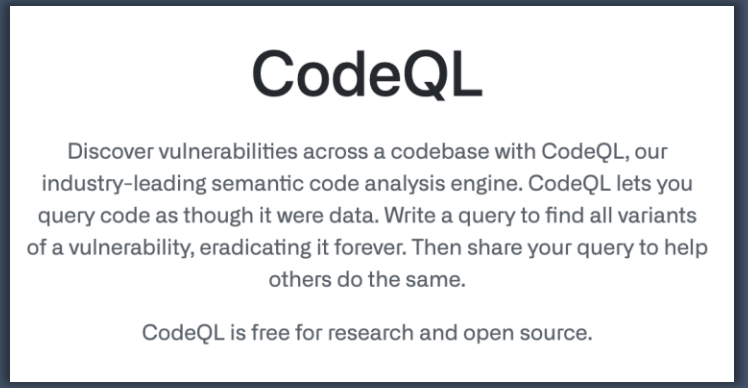
The key-value pairs have a consistent order that is determined by the sequence of insertion and removal calls on the map. The order does not depend on the keys or the hash function at all.

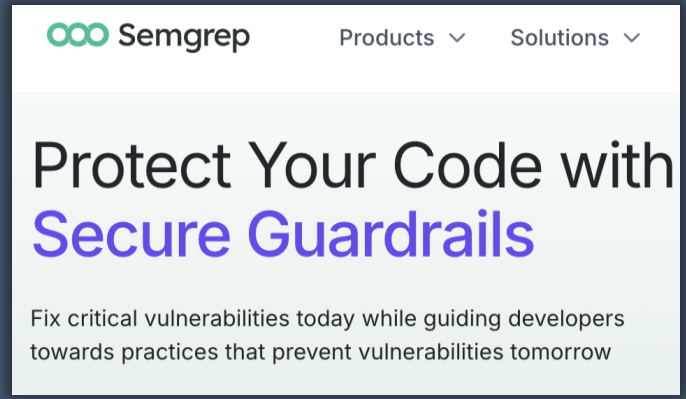
All iterators traverse the map in *the order*.

The insertion order is preserved, with **notable exceptions** like the `.remove()` or `.swap_remove()` methods. Methods such as `.sort_by()` of course result in a new order, depending on the sorting order.

How to avoid bugs?

- Tests, lots of them
- Static analysis
  - CodeQL: semantic code analysis engine “lets you query code as though it were data”
    - (pros) powerful, integrated into CI/CD pipelines
    - (cons) steep learning curve, need code to build
  - Semgrep: static analysis tool for locating specific code patterns
- Experience and careful review





Conclusion

- Presented common vulnerabilities in modern programming languages
- Validate all inputs!
- Test! Test! Test! (unit, integration/end-to-end, fuzzing)
- Hire an additional pair of eyes

paul.bottinelli@nccgroup.com