Enforcing Fresh Nonces with Affine Type System

Richard Ostertág

Department of Computer Science, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia

Abstract

Cryptographic algorithms and protocols often need fresh random numbers as parameters (e.g. nonces). Failure to satisfy this requirement lead to vulnerable implementation and can result in security breach. We show how affine type systems and static type checking can be used to enforce the correct generation of a new fresh random number for each function invocation.

Keywords

Secure coding, Nonce, Static checking, Substructural type systems, Rust

1. Introduction

The security of various cryptographic constructions relies on unique or even unpredictable values. Examples include nonces in cryptographic protocols, initialization vectors in modes of symmetric encryption, salts in password-based key derivation functions and others. These values are often generated as a random numbers of prescribed length.

Programmers who are not experts in cryptography might assume that it is not strictly necessary to generate a new random number every time. Some of them may be lazy and provide fixed numeric constant instead of a new random number for each use. After all, the cryptographic construction will “correctly”[1] work even with this fixed numeric constant. However, if the no-reuse principle is not followed, it can lead to a serious security vulnerability in the resulting application (which is not visible at first glance). A well-known example of this issue is “forbidden attack” for AES-GCM [1], but e.g. see also [2].

In this paper, we propose a method which demonstrate how to implement a cryptographic library that would allow the compiler to detect incorrect (i.e. repeated) use of such one-time random numbers at compile time. We will divide this task into two main parts:

1. *Ensuring Proper Random Number Generation:*
   In the first part, we ensure that the function expecting a random number gets as an argument a random number generated by an “approved” method. E.g. a true random number generated by a specialized hardware device and not just some pseudorandom number generated by weak algorithm. Alternatively, we can enforce the use of a specific, vetted software implementation.
   For this first part, we will employ abstract data types with hidden data constructors, to ensure that only approved random number generators can produce nonce values.
2. *Preventing Reuse of Random Numbers:*
   In the second part, we will ensure that once the generated random number is used, it cannot be reused for the second time.
   Substructural type systems which enforce single (or at most single) use semantics on values, can be utilized for this purpose. This approach can be applied not only to languages that support linear (single use) types but also to any language with similar features, such as ownership and borrowing in Rust, affine types in Haskell, or uniqueness types in Clean. We will illustrate this concept using the Rust programming language.

In Section 2, we describe the key features of abstract data types necessary to enforce proper random number generation. Next, in Section 3, we introduce the fundamental concepts of substructural type systems, focusing on linear and affine type systems that provide strict control over resource usage, which we utilize to prevent the reuse of random numbers. In Section 4, we then explore the unique features of the Rust programming language, including traits, ownership, move semantics, and borrowing rules, which we leverage to implement the nonce module. Finally, in Section 5, we demonstrate how to use the nonce module in a cryptographic library to enforce correct nonce usage.

2. Abstract data types

Abstract data types (in short ADTs) serve as a fundamental abstraction mechanism in computer science, providing a formal specification for data types that decouples their behavior from their concrete implementation. ADTs are defined by their external behavior, such as operations like

ITAT’24: Information technologies – Applications and Theory, September 20–24, 2024, Drienica, Slovakia

richard.ostertag@fmph.uniba.sk (R. Ostertág)
0000-0002-6560-1515 (R. Ostertág)

[1] Depending on the cryptographic construction, it might (for example) still correctly encrypt and decrypt messages.

insertion and deletion, while concealing the underlying implementation details from the user. This encapsulation grants implementers the flexibility to employ any internal data structures or modify their approach in the future. As long as the external behavior (interface) remains consistent, existing code utilizing the ADT will continue to function without requiring modifications to accommodate changes in the internal implementation. In this way ADTs also promote software reuse and modularity.

ADTs are widely used and supported in many standard programming languages, including C++, Java, or Pascal. They are typically realised as modules or objects that hide internal implementation details and expose only the public interface to the client. For instance, if we want to implement a stack (a LIFO data structure) as an ADT, we would provide public functions such as `push`, `pop`, and others and a type `Stack` for variables holding values of this ADT. But the important aspect is, that we do not disclose to the client any information on how the stack is internally implemented. It could be a linked list, an array, or something totally different. For example, if a more efficient data structure becomes available, the internal implementation of the ADT can be updated without affecting the client code, as long as the public interface remains unchanged. Additionally, we also do not provide any external means for creating a new stack (since external users lack knowledge of the internal details of the `Stack` type). The only way to create a new stack is to call a function from the module, which returns a new `Stack` value (or to create a new instance if objects are used instead of modules).

ADT are particularly useful for constraining access and preventing invalid states. By defining the stack as ADT, the module implementer can maintain strict control over its representation. Clients cannot accidentally or intentionally alter any of the stack's representation invariants. This ensures that the stack remains in a valid state, and its operations behave as expected. Therefore ADTs enhance code maintainability and readability.

We can use this technique to create a `nonce` module in the Rust programming language with `Nonce` abstract data type (see Listing 1).

We have defined a public struct type `Nonce` in Rust, encapsulating a private random value of type `u128`. Direct instantiation of structs with private field is prohibited. In this case for instance it is invalid to write `let nonce = Nonce { val: 42 }`. The only way for the client to create a nonce is to invoke a public constructor method like `let mut nonce = nonce::Nonce::new()`.

Since the client needs to call the `new` method, we can ensure that on line 12, we, as implementers, select the appropriate system function to generate a new random number. This could potentially involve using a hardware RNG for added security. However, to keep this example simple, this step is not included.

```
1  mod nonce {
2    // A public struct with a private
3    // random value of type u128
4    pub struct Nonce {
5      val: u128,
6    }
7
8    impl Nonce {
9      // A public constructor method
10     pub fn new() -> Nonce {
11       use rand::prelude::*;
12       Nonce { val: random() }
13     }
14
15     // A public getter method
16     pub fn get(self: &Self) -> &u128 {
17       &self.val
18     }
19   }
20
21   // The Copy and Clone traits are
22   // intentionally not implemented
23
24   // DerefMut is needed to modify through
25   // a dereference, so since only Deref
26   // is defined, nonce cannot be modified
27   use std::ops::Deref;
28   impl Deref for Nonce {
29     type Target = u128;
30     fn deref(self: &Self) -> &u128 {
31       &self.val
32     }
33   }
34 }
```

Listing 1: Implementation of nonce module in Rust

While abstract types are a powerful means of controlling the structure and creation of data, they are not sufficient to limit the ordering and number of uses of values and functions. As another example, we can mention e.g. files. There is no (static) way to prevent a file from being read after it has been closed [3] utilising only ADTs. Additionally, it is challenging to enforce rules preventing clients from erroneously closing files multiple times or forgetting to close them altogether. Similarly, with our `Nonce` example, there is no static way to stop the client from using one nonce value multiple times just by using ADTs alone. However, this can be enforced in programming languages that support the appropriate substructural type system.

3. Substructural type systems

Before presenting our proposed solution using substructural type system, it's essential to provide a brief overview of what substructural type systems are [3] and how they are implemented within the well-known Rust programming language [4].

Linear and affine type systems are a special case of substructural type systems. They are particularly beneficial in scenarios where strict control over resource usage is crucial and we need a way for constraining usage of the interface of this resource (imagine resources like files or memory). By utilizing linear (or affine) types, we can ensure that certain values or operations are used exactly (or at most) once and in some way in the correct sequence, thereby preventing common programming errors and enhancing the security and reliability of software systems.

In the context of our forthcoming solution, we will demonstrate how substructural type system in Rust enable us to enforce the one-time use of random numbers, thereby mitigating potential security vulnerabilities associated with nonce reuse in cryptographic applications. This approach not only leverages Rust's robust type system but also showcases the practical application of advanced type theories in real-world software development.

3.1. Structural Properties

In accordance with Pierce's work [3] based on simply-typed lambda calculus, we will treat the type-checking context, denoted as Γ, as a straightforward list of variable-type pairs, $x : \tau_x$, where x represents a variable and τ_x denotes its type.

The comma operator (,) serves to append a pair to the end of the type-checking context (e.g. $\Gamma_1, x : \tau_x$), or to concatenate two type-checking contexts (e.g. Γ_1, Γ_2). Let us denote by $\Gamma \vdash e : \tau$ that, within the context Γ, we can type-check that the term e has the type τ.

We denote the substitution of the term y for the free variable x in the term e by $[x \mapsto y]e$. We assume that x and y have the same type, ensuring that the resulting term $[x \mapsto y]e$ is also correctly typed (in simply-typed lambda calculus). Instead of writing $[x_3 \mapsto x_4]([x_1 \mapsto x_2]e)$, we use the more concise form $[x_1 \mapsto x_2, x_3 \mapsto x_4]e$.

$$\frac{\text{premise}}{\text{conclusion}} \qquad \text{(Typing rule)}$$

Finally, by typing rule we mean that if its premise is true, then we can conclude that its conclusion also holds.

Lets discuss three basic *structural* properties. The first property, *exchange*, indicates that the order in which we write down variables in the context is irrelevant. A corollary of exchange is that if we can type check a term with the context Γ, then we can type check that term with any permutation of the variables in Γ.

$$\frac{\overbrace{\Gamma_1, x : \tau_x, y : \tau_y, \Gamma_2}^{\text{context}} \vdash e : \tau}{\underbrace{\Gamma_1, y : \tau_y, x : \tau_x, \Gamma_2}_{\text{permutated context}} \vdash e : \tau} \qquad \text{(Exchange)}$$

The second property, *weakening*, indicates that adding extra, unneeded assumptions to the context, does not prevent a term from type checking.

$$\frac{\Gamma \vdash e : \tau}{\Gamma, \underbrace{x : \tau_x}_{\text{unneeded assumption}} \vdash e : \tau} \qquad \text{(Weakening)}$$

Finally, the third property, *contraction*, states that if we can type check a term using two identical assumptions ($x_2 : \tau_x$ and $x_3 : \tau_x$) then we can check the same term using a single assumption.

$$\frac{\Gamma, x_2 : \tau_x, x_3 : \tau_x \vdash e : \tau}{\Gamma, x_1 : \tau_x \vdash [x_2 \mapsto x_1, x_3 \mapsto x_1]e : \tau} \qquad \text{(Contraction)}$$

In his book [3], Pierce employs simply-typed lambda calculus as a foundation to introduce the concepts of linear and ordered lambda calculus. While the intricate details of these theoretical underpinnings are beyond the scope of our paper, we encourage interested readers to consult Pierce's work for a comprehensive exploration. His book provides an in-depth discussion on the principles of substructural type systems and their applications, making it an valuable resource for those looking to delve deeper into this topic.

3.2. Substructural Type Systems

A *substructural type system* is any type system that is designed so that one or more of the structural properties do not hold [3]. Different substructural type systems arise when different properties are withheld.

Linear type systems ensure that every variable is used exactly once by allowing exchange but not weakening or contraction.

Affine type systems ensure that every variable is used at most once by allowing exchange and weakening, but not contraction.

Relevant type systems ensure that every variable is used at least once by allowing exchange and contraction, but not weakening.

Ordered type systems ensure that every variable is used exactly once and in the order in which it is introduced. Ordered type systems do not allow any of the structural properties.

The Fig. 1 below[2] can serve as a mnemonic for the relationship between these systems. The system at the bottom of the diagram (the ordered type system) admits no structural properties. As we proceed upwards in the diagram, we add structural properties: E stands for exchange; W stands for weakening; and C stands for contraction.

$$\begin{tikzcd}
& \text{unrestricted (E,W,C} \Rightarrow \text{structural)} & \\
\text{affine (E,W)} \arrow[ur, "+C"] & & \text{relevant (E,C)} \arrow[ul, "+W"'] \\
& \textbf{linear (E)} \arrow[ul, "+W"] \arrow[ur, "+C"'] & \\
& \text{ordered (none)} \arrow[u, "+E"'] &
\end{tikzcd}$$

Figure 1: Relationship between linear and other substructural type systems.

All type systems below unrestricted (aka. structural) are called substructural type systems. It might be possible to define type systems containing other combinations of structural properties, such as contraction only or weakening only, but so far researchers have not found applications for such combinations [3]. Consequently, they are excluded from the diagram.

4. Rust

Ownership is Rust's most unique feature which is closely related to the concept of substructural type systems in that both systems enforce strict rules about how data is accessed and modified to ensure safety and correctness. It enables Rust to make memory safety guarantees without needing a garbage collector. In Rust, the memory is managed through a system of ownership with a set of rules, that the compiler checks at compile time. None of the ownership features slow down the program while it is running (unlike garbage collection).

4.1. Rust traits

Traits in Rust serve as a way to define shared behavior. They are similar to interfaces in other programming languages. They allow to specify methods that types must implement, enabling polymorphism and code reuse. For more see e.g. [5].

In Rust, there are several important standard traits that provide foundational functionality and are widely used. For the purposes of this paper, it is important to understand, that the `Copy` and `Clone` traits allow for duplication of value. Therefore, we do not implement them for the `Nonce`. The `Deref` trait only allows for reading a stored value. To enable writing, the `DerefMut` trait is necessary, which we also do not implement. Programmers can access the stored `u128` value using the `nonce.get()` method or by dereferencing with `*nonce`. Both return an immutable reference (as in immutable borrowing).

4.2. Ownership rules

In Rust, when a value is assigned to a variable, the variable becomes the "owner" of that value. When the owner variable goes out of scope, the value is automatically deallocated. Rust enforces that there can only be one owner for a value at a time. This is the essence of substructural type system in Rust [6]:

- Each value has a variable that is called its *owner*.
- There can be only one owner at a time.
- When the owner goes out of scope, the value will be dropped (memory will be deallocated).

4.3. Move semantics

Move semantics in Rust is a fundamental concept that allows the transfer of ownership of a value from one variable to another. When a value is assigned from one variable to another (e.g. `let s2 = s1;`) or passed to a function, ownership of the value is transferred (aka moved) from `s1` to the new variable `s2` unless the value implements the `Copy` trait [7]. In other words, variable bindings have "move semantics" if their type does not implement the `Copy` trait; otherwise, they have "copy semantics".

After a move, the original variable is no longer valid and cannot be used. Move semantics can improve performance by avoiding deep copies of data. Instead of copying the data, Rust only copies the pointer to the data and invalidates the original pointer. Single owner allows for values to be deallocated as soon as their owner goes out of the scope.

4.4. Borrowing rules

If you need to access or modify a value without transferring ownership, you can borrow a reference to it. There are two types of borrowing in Rust:

- *Immutable borrowing:*
  You can have multiple immutable references to a value, but you cannot modify the value through

[2] Fig. 1 is based on similiar image in [3].

these references. This prevents data races because multiple threads can read a value without the risk of it being modified simultaneously.

- *Mutable borrowing:*
  You can have only one mutable reference to a value, and no other references (mutable or immutable) can coexist with it. This enforces exclusive access to the value, ensuring that only one part of the code can modify it at a time.

```
1  fn borrowing() {
2    let s1 = String::from("Hello");
3    //  ^^ move occurs because `String` does
4    //     not implement the `Copy` trait
5
6    let s2 = s1; // value moved from s1 to s2
7
8    println!("{}, world!", s1);
9    //error: val. borrowed ^^ here after move
10 }
```

Listing 2: Classical example of ownership rules

We will demonstrate some of these rules on Listing 2. On line 2 we create a string and assign its value into variable `s1`. This variable is now the only owner of the string. Then on line 6 we move value from variable `s1` to new owner – variable `s2` (because `String` does not implement `Copy` trait). Now `s2` is the only owner of the string value. That is the reason, why we can not use variable `s1` on line 8 to borrow the string value to `println!` function. But we could use `s2` for this. When `s2` comes out of scope the string value can be deallocated from memory. This is illustrated in Fig. 2 on the right[3]. After assigning to `s2` the value from `s1`, variable `s2` points to the same memory on the heap, but `s1` can not be used for dereferencing anymore. This is used primarily for memory management without the need for a garbage collector or explicit deallocation. For more see e.g. [6].

5. The solution

The solution in Rust is syntactically very simple because it is well aligned with Rust syntax. Usually, when functions in Rust take arguments, they are passed as references (with `&` before variable name). This way value is not moved to the parameter from the local variable (it is just borrowed). However, we can prevent borrowing by not taking reference as the argument and not implementing `Copy` trait in `Nonce` type.

[3]Fig. 2 is based on similiar image in [6].

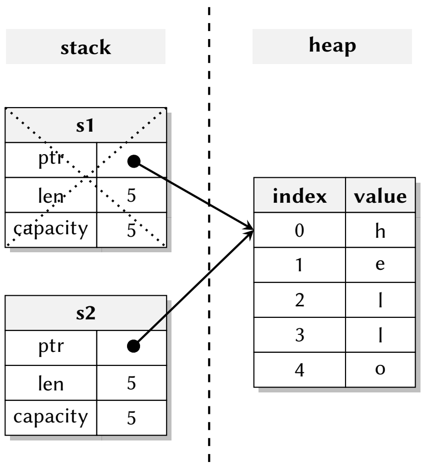

Figure 2: Memory representation of variables `s1` and `s2`

On Listing 3 we implement function `need_new_random_u128_every_time` to demonstrate function signature for functions that require fresh random value for every call. The body of the function is not significant, but we demonstrate, that the nonce value can be used repeatedly inside library implementation, which is often needed. We also implement `Deref` trait, so `*` can be used on line 10 instead of longer `nonce.get()` from line 7.

```
1  fn need_new_random_u128_every_time(
2    nonce: nonce::Nonce
3  ) {
4    let _tmp = nonce.get();
5
6    println!("Nonce param value: {}",
7      nonce.get());
8
9    println!("Nonce param value: {}",
10     *nonce);
11 }
```

Listing 3: Example of function with nonce as argument

When function `need_new_random_u128_every_time` is called, then value ownership is moved from the local variable to the argument and thus local variable can not be used anymore. As an example, if in Listing 4 we comment out line 7, we will get compile time error “value used here after move” on the next line.

6. Conclusion

We have demonstrated how to use abstract data types and substructural type systems for enforcing the freshness of

```rust
fn main() {
  // Structs with private fields can be
  // created only using public constructors
  let mut nonce = nonce::Nonce::new();
  need_new_random_u128_every_time(nonce);

  nonce = nonce::Nonce::new();
  need_new_random_u128_every_time(nonce);

  need_new_random_u128_every_time(
    nonce::Nonce::new()
  );
}
```

Listing 4: Example of nonce usage

nonces for cryptographic library function calls. In Rust, the syntax is very straightforward. This solution can be implemented also in other languages with affine or linear type system, like Haskell, which experimentally supports linear types from version 9.0.1. But syntax, in this case, is not so clear as in Rust.

Acknowledgments

This publication is the result of support under the Operational Program Integrated Infrastructure for the project: Advancing University Capacity and Competence in Research, Development a Innovation (ACCORD, ITMS2014+:313021X329), co-financed by the European Regional Development Fund.

References

[1] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, P. Jovanovic, Nonce-Disrespecting adversaries: Practical forgery attacks on GCM in TLS, in: 10th USENIX Workshop on Offensive Technologies (WOOT 16), USENIX Association, Austin, TX, 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/bock.

[2] A. Joux, Authentication failures in NIST version of GCM, 2006. URL: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf.

[3] B. C. Pierce (Ed.), Advanced topics in types and programming languages, The MIT Press, MIT Press, London, England, 2004.

[4] S. Klabnik, C. Nichols, The Rust programming language, No Starch Press, San Francisco, CA, 2019.

[5] S. Klabnik, C. Nichols, Traits: Defining shared behavior, 2024. URL: https://doc.rust-lang.org/book/ch10-02-traits.html, with contributions from the Rust Community.

[6] S. Klabnik, C. Nichols, What is ownership?, 2024. URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html, with contributions from the Rust Community.

[7] S. Klabnik, C. Nichols, Trait std::marker::Copy, 2024. URL: https://doc.rust-lang.org/std/marker/trait.Copy.html, with contributions from the Rust Community.