Proving Secrets' Safety: Verifying Secure Information Flow in a Rust Verifier

Master Thesis Project Description

Alexandre Pinazza
Supervised by Aurel Bílý*, Prof. Dr. Peter Müller* and Prof. Dr.Viktor Kunčak†
*Department of Computer Science
ETH Zürich
Zürich, Switzerland
†LARA, EPFL
Lausanne, Switzerland

I. Introduction

Rust is a multi-paradigm programming language with a focus on performance and safety. It is designed to be suitable for low- and high-level programming and can be used to program drivers, operating system kernels, embedded systems, front- and back-end applications, and much more. The unique feature of Rust is its ownership type system for automatic memory management and safety. It prevents memory corruption and data races.

Using Rust's powerful ownership and type system, Astrauskas et al. [1] introduced *Prusti*, a formal verification tool to verify Rust programs using the *Viper* verifier [2].

Rust's high safety standards and performance make it a compelling programming language to implement security protocols. To this end, this work aims to extend Prusti's capabilities to support the detection of secret information leaks, such as through timing attacks or other channels.

In this project, we will consider two security levels: *high* and *low*, and we want to guarantee the safety of high-security values against attackers who can: control low-security inputs, observe low-security outputs, and also measure the execution time of functions. To achieve it, we will first add specifications for the functions' runtime and create a runtime model to prove them. Then we will extend Prusti with the ability to check for noninterference as done by Eilers et al. [3]. Finally, we will combine the two to check that the execution time does not leak high-security information thus making the function timing-attacks proof.

II. Security Properties

A. Timing attacks

Timing attacks are possible when a function's runtime depends on a secret's value. Attackers can perform them remotely if the delays of the network connection are not significant. For example, let's consider a function that compares two same-size arrays. An optimized implementation would be as in Listing 1. This implementation can be insecure, as its (non-asymptotic) runtime can leak information about where the two arrays differ, even though the

```rust
#[requires(a1.len() == a2.len())]
fn opt_array_cmp<T: Eq>(a1: &[T], a2: &[T]) -> bool {
    let mut i = 0;
    while i < a1.len() {
        if a1[i] != a2[i] {
            return false;
        }
        i += 1;
    }
    return true;
}
```

Listing 1: An array comparison implementation with an early return.

```rust
#[requires(a1.len() == a2.len())]
fn safe_array_cmp<T: Eq>(a1: &[T], a2: &[T]) -> bool {
    let mut i = 0;
    let mut res = true;
    while i < a1.len() {
        res = res && a1[i] != a2[i];
        i += 1;
    }
    return res;
}
```

Listing 2: A safe array compare implementation.

function has an asymptotic runtime of $O(a1.len())$. This is because as soon as the loop finds different elements, it returns `false`. So if the first different elements are at the beginning or the end of the arrays, the function's runtime will differ. Thus, if this function is used to compare a user input and a high-security array, an attacker, by measuring the response time of the function, might guess the values of the secret array quicker than using brute force.

Safe implementations of this function would have a runtime proportional to the public argument as in Listing 2. This implementation always compares all the elements and thus its runtime is always the same for given-size arrays, so it does not leak information. However, note that even though it is written to prevent timing attacks, it does not guarantee that the compiler does not perform some optimizations that introduce such vulnerabilities. For example, this can happen if the compiler partially unrolls the loop.

Additionally, note that we will not take into account changes in timing due to cache or pipelining. Consequently, our procedure will not be perfect. For example, the code in Listing 3, adapted from [4], executes faster on processors with an L1 data cache of 16KB when `secret` is `true` compared to when it is `false`. It happens because, in the first case, the function reads only from array `a1` which fully fits in the L1 cache. In the other case, the function reads from both arrays, and as both arrays cannot fit into the L1 cache some of the data have to be evicted at each iteration of the outer loop causing cache misses.

```rust
fn fancy_sum(secret: bool, loop_count: u32,
             a1: [u64; 4000], a2: [u64; 4000],
) -> u64 {
    let mut l = 0;
    let mut res = 0;
    while l < loop_count {
        let mut i = 0;
        while i < 4000 {
            res += a1[i];
            if secret {
                res += a1[i];
            } else {
                res += a2[i];
            }
            i += 1;
        }
        l += 1;
    }
    res
}
```

Listing 3: Rust function whose runtime depends on the size of the L1 cache.

```rust
fn foo(secret: u32) {
    if secret % 2 == 0 {
        println!("bar");
    } else {
        println!("baz");
    }
}
```

Listing 4: Function leaking 1 bit of information through the console.

B. *Secure Information flow*

Sometimes information leaks through media that are not secure, such as writing into a file, onto the network, or writing to the console. To prevent this, we must check for noninterference, which means that no variable holding a high-security value influences the value of any low-security expressions. Listing 4 shows an example that leaks the last bit of the secret argument to the console.

By default, all variables are high-security and the users need to specify which expressions are low. This has two advantages, the first one is that information is secured by default preventing some underspecification mistakes. Secondly, it allows the use of arbitrary expressions and not simply variables in the `low` specification as in Listing 5. Additionally, this example shows that `low` specifications are usable in implications.

To prevent information leaks, high-security values should not influence `low` expressions. However, we sometimes need to declassify secrets into low-security variables. For example, the output of a password-checking function or an encryption function's output. For those cases,

```rust
#[requires(low(i1))]
#[ensures(b ==> low(result % 2))]
fn choose(b: bool, i1: i32, i2: i32) -> i32 {
    if b {
        i2 * 2 + i1 % 2
    } else {
        i2
    }
}

#[ensures(low(i) ==> low(result)]
fn add_one(i: i32) -> i32 {
    i + 1
}
```

Listing 5: Rust `low` specification syntax.

```rust
#[trusted]
#[ensures(low(result))]
fn declassify<T>(t: T) -> T {
    t
}
```

Listing 6: Declassify function in Rust.

the users call a `declassify` function. Such a function can be defined as in Listing 6. It needs to be `#[trusted]` as it does not satisfy its specification.

III. Approach

A. Timing attacks

Preventing timing attacks requires proving that the functions' runtime does not depend on high-security values we wish to preserve. Engel [5] showed how to reason about asymptotic upper bound time complexities in Prusti. In this project, we do not only want to reason about upper bounds but also lower bounds. In addition, asymptotic time is not precise enough as the constant might hide our secrets. For example, we might have a function that has a runtime of $\Theta(n)$ where n is the length of a non-secret array. Still, in reality, the high-security data might change the runtime between n and $2n$, as in Listing 7, which would be detectable in a timing attack.

To reason about upper bounds, Engel used *time credits* [6], [7] to model a function's runtime as a resource with each operation consuming some amount of this resource. To prove the correctness of the runtime, we need to verify that at every step in the execution of the function, the time credit is greater or equal to zero. To reason about lower bounds, we will reason about *time receipts*, which guarantees that a function will produce at least this amount of time resource by the end of its executions. Listing 8 shows one way to encode them in Viper. We also must check that no time resources are leaked in variables or fields.

We can apply this reasoning to a function summing the elements of an integer array. The Rust implementation can be found in Listing 9 and its approximate translation to Viper in Listing 10.

```rust
fn sum(array: &[u32], secret: bool) -> u32 {
    let mut res = 0;
    let mut i = 0;
    while i < array.len() {
        res += array[i];
        if secret {
            res += array[i];
        }
        i += 1;
    }
    res
}
```

Listing 7: $\Theta(n)$ function which leaks secret.

```
predicate time_credits()
predicate time_receipts()

method tick(n: Int)
    requires 0 <= n
    requires acc(time_credits(), n/1)
    ensures acc(time_receipts(), n/1)
```

Listing 8: Time credits and receipts encoding in Viper.

```rust
#[requires(time_credits(5 * a.len() + 2))]
#[ensures(time_receipts(5 * a.len() + 2))]
fn array_sum(a: &[i32]) -> i32 {
    let mut i = 0;
    let mut res = 0;
    while i < a.len() {
        body_invariant!(0 <= i && i <= a.len());
        body_invariant!(time_credits(5 * a.len() - 5 * i));
        body_invariant!(time_receipts(5 * i));
        res += a[i];
        i += 1;
    }
    res
}
```

Listing 9: Rust code checking lower and upper bound.

```
method array_sum(a: Seq[Int]) returns (res: Int)
  requires acc(time_credits(), (5 * |a| + 2)/1)
  ensures acc(time_receipts(), (5 * |a| + 2)/1)
{
  tick(2) // initialize i and res
  var i: Int := 0
  res := 0
  while (i < |a|)
    invariant 0 <= i && i <= |a|
    invariant acc(time_credits(), (5 * |a| - 5 * i)/1)
    invariant acc(time_receipts(), (5 * i)/1)
  {
    tick(2) // loop condition and jump
    tick(3) // two additions and one memory read
    res := res + a[i]
    i := i + 1
  }
}
```

Listing 10: Viper code checking lower and upper bound.

B. *Secure Information Flow*

Checking for noninterference can be done using type systems [4]. It might also be done using Rust's powerful type system. For example, using a generic type `Secret<T>` or `Low<T>` for each type `T`. But using this approach would prevent calling `T`'s methods directly on their security-aware variants. A workaround for this issue would be to implement the `Deref<T>` trait on those generic types. However, this causes other issues that we'll showcase with the example in Listing 11. Implementing `Deref` on `Secret<T>` is dangerous as the compiler could declassify any high-security value (for example in line 26) without requiring an explicit action from the user. Dually, implementing it on `Low<T>` would allow the compiler to classify low-security values (at line 29, the result of `l.plus1()` is not a generic type so it is a high-security value) which requires the users to declassify them to make them low-security.

Another approach is to use a 2-hyperproperty saying that two executions of a program with the same value for each low-security input variable yield the same values for each low-security output variable. Eilers et al. showed how to check such properties using modular product programs in Viper [8] and used it in the *Nagini* verifier to check secure information flow in Python programs [3].

Using product programs allows the representation of two different executions of the same function as a normal one. This, in turn, allows the encoding of 2-hyperproperties as a regular assertion. In this encoding, the `low` assertion translates into the equality of the given expression in each of the two executions.

However, Eilers' implementation represents programs as *abstract syntax trees* – with ifs and while – whereas, Prusti uses *control flow graphs*. Fortunately, Prusti's CFG only contains forward jumps as Prusti encodes loops before exporting programs to Viper. So we should be able to translate it into an AST for Viper. If it is not possible, either we translate the programs into their modular products form in Prusti or we extend Eilers' transformation to

```rust
struct Secret<T>(T);
impl<T> Deref for Secret<T> {
    type Target = T;
    fn deref(&self) -> &self.Target {
        &self.0
    }
}

struct Low<T>(T);
impl<T> Deref for Low<T> {
    type Target = T;
    fn deref(&self) -> &self.Target {
        &self.0
    }
}

struct Myi32(i32);
impl Myi32 {
    fn plus1(&self) -> i32 {
        self.0 + 1
    }
}

fn main() {
    let s = Secret(Myi32(1));
    println!("{}", s.plus1());

    let l = Low(Myi32(2));
    println!("{}", l.plus1());
}
```

Listing 11: Deref on `Seret` and `Low` type

handle CFG. The former would be ideal as it would enable other Viper's front ends to use this feature, but it would require modifying Viper's codebase in addition to Prusti's, which might be out of this project's scope. So if we need to implement a new transformation, where we will implement it will depend on the available time. We must also be careful and check that Prusti's transformations are sound with the product transformation. If the transformation does not work, we will try other approaches such as using Rust's borrow checker proposed by Crichton et al. [9].

C. Combining everything

In Section III-A, we explained how we want to check the timing of functions in a fine-grain manner. The final step to prevent timing attacks is to check that the timing expression does not leak any secrets. We can use the same procedure as in Section III-B on the timing expressions.

Eilers et al. [8] explained how to prevent timing attacks directly in the secure information flow check, but splitting it in two could give us more fine-grain control and also a performance gain as we only need to perform the program product on the timing expressions rather than on the entire function.

IV. Core Goals

The following list contains the core goals and their estimated time.

- **Adding time credits, receipts syntax**
  In this goal, we'll add the assertion syntax for specifications of time credits and receipts.
  Estimated time: 2 weeks
- **Create a model for the resources needed for each operation in Rust**
  This goal aims to implement a model for the timing of the Rust expressions handled by Prusti and the generation of the required Viper code.
  Estimated time: 2 weeks
- **Fine-tuning and evaluation of the model**
  This goal is to write examples and evaluate the correctness of the model.
  Estimated time: 1 week.
- **Adding `low` syntax and encoding, and transforming Prusti's CFG into AST for SIF or reimplementing the modular product program**
  We can't use Viper's SIF extension as it is, because it is incompatible with Prusti's program representation. So in this goal, we need to transform it into a compatible representation for Viper.
  Estimated time: 4 weeks.
- **Using noninterference on timing to check that no information is leaked through the function's runtime**
  This goal combines the implementation of the time credits and receipts with the secure information flow to check for information leaks in the runtime of functions.
  Estimated time: 1 week.
- **Evaluation of the secure information flow**
  Writing examples and assertion of the secure information flow check mechanism.
  Estimated time: 1 week.
- **Intermediate report writing**
  Writing of the intermediate report.
  Estimated time: 2 weeks.
- **Final thesis writing**
  Writing of the final thesis report and preparation for the final project presentation.
  Estimated time: 4 weeks.

V. Extension Goals

The estimated time allocated for the extension goals is 3 weeks.

- **Make it simpler to figure out the constants in the time credits and receipts annotations**
  As it is, specifying the time credits and time receipts constraints requires knowing the underlying model Prusti uses which is not user-friendly. This goal is to find a way to simplify this procedure.

- **Add syntax to define the upper and lower bounds at once**
  In constant time functions, the upper and lower bounds should be the same so in those cases, writing only one specification would be more user-friendly.
- **Add warning if `time_credits` are used in post-conditions and `time_receipts` in pre-conditions** As presented in Section III-A, `time_credits` should be used in preconditions and `time_receipts` in post-conditions. However, we don't plan to implement a mechanism preventing the use of `time_credits` in post-conditions and `time_receipts` in pre-conditions. So in this goal, we want to add a warning to point this out to the user.

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, Oct. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3360573

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, vol. 9583, pp. 41–62, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-662-49122-5_2

[3] M. Eilers, S. Meier, and P. Müller, "Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, vol. 12759, pp. 718–741, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-81685-8_34

[4] G. Smith, "Principles of Secure Information Flow Analysis," in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA: Springer US, 2007, vol. 27, pp. 291–307, series Title: Advances in Information Security. [Online]. Available: http://link.springer.com/10.1007/978-0-387-44599-1_13

[5] L. Engel, "Reasoning about Complexities in a Rust Verifier," p. 79.

[6] A. Charguéraud and F. Pottier, "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits," *Journal of Automated Reasoning*, vol. 62, no. 3, pp. 331–365, Mar. 2019. [Online]. Available: http://link.springer.com/10.1007/s10817-017-9431-7

[7] G. Mével, J.-H. Jourdan, and F. Pottier, "Time Credits and Time Receipts in Iris," in *Programming Languages and Systems*, L. Caires, Ed. Cham: Springer International Publishing, 2019, vol. 11423, pp. 3–29, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-17184-1_1

[8] M. Eilers, P. Müller, and S. Hitz, "Modular Product Programs," *ACM Transactions on Programming Languages and Systems*, vol. 42, no. 1, pp. 1–37, Mar. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3324783

[9] W. Crichton, M. Patrignani, M. Agrawala, and P. Hanrahan, "Modular Information Flow through Ownership," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Jun. 2022, pp. 1–14, arXiv:2111.13662 [cs]. [Online]. Available: http://arxiv.org/abs/2111.13662