Automated verification of a Rust differential privacy library

Master Thesis Project Description

Till Arnold
Supervised by Jonáš Fiala, Anouk Paradis, Prof. Dr. Peter Müller
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Introduction and Background

Rust [1] is a system programming language which guarantees memory safety. While Rust eliminates common classes of bugs and security vulnerabilities such as dangling pointers, data races and buffer overruns, it does not ensure the general functional correctness of programs. Functional correctness can be checked by static verifiers, which traditionally, especially for system languages, require a formal verification expert to provide complex program specifications.

Prusti [2] is a verifier for Rust which allows non-expert users, such as programmers, to provide simpler specifications, syntactically close to regular Rust code. Internally Prusti translates Rust code into the Viper [3] intermediate verification language. The functional correctness of the Rust program then follows from the proof that the translated Viper program is correct. Listing 1 shows a simple example of a Rust program annotated with Prusti attributes.

Differential privacy (DP) describes an approach which allows for the computation of statistics while preserving the privacy of individuals who contributed to the input dataset. This is accomplished by introducing noise to the result in such a way that, for any two datasets which are "close" to each other, for some notion of closeness, such as for example the datasets only differing by the answers of a single individual, the results are approximately the same. This inherently reduces the accuracy of the statistical results. However, differentially private algorithms are able to explicitly calculate this reduction in accuracy as well as the amount of privacy which is afforded to each individual and can often be parameterized to prioritize one over the other.

```
1  struct Account { bal: u32 }
2
3  impl Account {
4    #[pure]
5    fn balance(&self) -> u32 { self.bal }
6
7    #[ensures(self.balance() ==
8    old(self.balance()) + amount)]
9    fn deposit(&mut self, amount: u32) {
10     self.bal = self.bal + amount;
11   }
12
13   #[requires(amount <= self.balance())]
14   #[ensures(self.balance() ==
15     old(self.balance())  - amount)]
16   fn withdraw(&mut self, amount: u32) {
17     self.bal = self.bal - amount;
18   }
19
20   #[requires(amount <= self.balance())]
21   #[ensures(self.balance() ==
22     old(self.balance())  - amount)]
23   fn transfer(&mut self, other: &mut Account,
24   amount: u32) {
25     self.withdraw(amount);
26     other.deposit(amount);
27   }
28 }
29
```

Listing 1: Simple example for the attributes Prusti uses to annotate functions. This example is taken from `account.rs` from the Prusti tests.

OpenDP [4] is an open-source library of differentially private algorithms implemented in Rust. In broad terms, OpenDP works by constructing two kinds of operators: *transformations*, which are deterministic functions from datasets to datasets, and *measurements*, which are randomized functions from datasets to outputs of arbitrary types. Transformations have a *stability* parameter and measurements have a *privacy loss* parameter associated with it. More complex operators can be constructed by function composition of transformations with other operators. The stability and privacy loss for operators constructed by function composition can directly be computed from the stability and privacy loss of the underlying operators. The OpenDP library keeps track of the stability

and privacy loss as well as the input and output domains of each operator. Additionally a notion of closeness of datasets in the input and output domains, called a *metric*, is stored with the operator. OpenDP ensures that the domains and metrics are compatible when applying function composition, utilizing a mixture of compile time and runtime checks. [5]

II. Problem Statement

An explicitly stated goal of the OpenDP project is to "ensure that the only measurements and transformations that can be constructed by the OpenDP Library have mathematically proven privacy properties" [5]. For this purpose the OpenDP project contains textual proofs for some operators written as LaTeX documents stored alongside their respective Rust implementation. Since the correctness of the algorithms in OpenDP is of central importance, the OpenDP project proposes that proofs should be "verified by a human [...] or by a computer [...] for components that are amenable to formal verification techniques" [5]. Although there is currently no automatic verification and the project relies solely on verification by humans, it would be possible to leverage Prusti for this purpose, especially since OpenDP does not heavily utilize some more advanced Rust language features, such as complicated borrow structures, unsafe code or interior mutability, which are currently not supported within Prusti and would be difficult to implement. As such, OpenDP is good candidate for verification with Prusti.

Preliminary investigation of the existing LaTeX proofs indicates that the proven transformations share similar postconditions, in particular transformations are called *valid* if they fulfill the following properties:

- *Appropriate output domain*: If an element is in the input domain of a transformation, then applying this transformation to the element either results in an element of the output domain of the transformation or alternatively raises a runtime exception.
- *Stability guarantee*: For any pair of elements u and v that are `d_in`-close under the input metric and if `stability(d_in)` $\leq$ `d_out`, the results of applying the transformation to u and v are `d_out`-close under the output metric. An analogous property called *privacy guarantee* exists for measurements.
- *Data-independent runtime exceptions*: The runtime errors raised by transformations have to be data-independent. This is to prevent leaking any information about the data, similarly to how this property would be desirable in a cryptography library.

The later two properties cannot simply be expressed as a regular postcondition of the function. For example in the case of the data-independent runtime exceptions we are trying to show that for two different executions, which share all the same parameters aside from the data, the resulting errors cannot differ. Such properties, which relate different executions of the same program, are referred to as hyperproperties and are well understood in formal verification, there is however currently no support for this in Prusti. Potential solutions include constructing a product program [6] or using information flow [7].

The function composition of operators which uphold these properties should result in operators which themselves also uphold the properties. Thus operators constructed by function composition of valid operators are themselves valid.[5]

Finally, while OpenDP is a good fit for verification with Prusti, many features necessary to verify the above properties are not currently implemented in Prusti.

III. Core Goals

The main goal of this thesis is to verify parts of the OpenDP Rust library and to implement the Prusti features needed to do so. For this purpose we define the following core goals:

1) **Decide on which parts of OpenDP to verify and identify the correspondingly necessary Prusti features.** Identify the fundamental and most useful operators within OpenDP, taking into account their dependences and preexisting proofs written by the OpenDP contributors.
   Operators with existing proofs are better candidates, not only because the known pre- and postconditions lend themselves to a simpler verification process, but also because more important transformations are likely to be have been prioritized by the OpenDP contributors.
   From our preliminary investigation of the OpenDP code, we selected the appropriate output domain and data-independent runtime exceptions properties as a focus for this thesis, based on both feasibility of implementation and usefulness to the verification process.
2) **Implement the Prusti features needed for the selected properties.** Implement the specifications for the properties and operators, and implement the Prusti features needed to complete the verification. This includes improving and extending existing features and making them more robust to allow for the verification of the selected properties to complete. This might potentially include closures, as they are contained within the transformation and measurement data structures. Closures in Prusti have been described and implemented before [8], but are not supported in the current Prusti version.
3) **Evaluate the work.** Qualitatively evaluate the work in regards to the number of verified functions, the importance of the verified properties to the OpenDP project, properties which cannot currently be verified and the reason why this is not feasible and whether or not actual privacy guarantees can be made. Include the verified functions in the Prusti tests.

IV. Extension Goals

1) **Stability Guarantee:** The stability guarantee property as described above is important, but requires significant implementation work in Prusti. Once the stability guarantee property for transformations can be verified using Prusti, it may become possible to show the analogous privacy guarantee property for measurements. One potential problem with the privacy guarantee property could be that measurements by their nature include randomness, which may require further reasoning.
2) **Additional properties:** The list of identified properties above might be non-exhaustive. If other relevant properties are found during the implementation, they could also be implemented.
3) **Verify more operators:** In addition to those operators with pre-existing LaTeX proofs, those without may still be verified although the necessary preconditions would have to be worked out first.
4) **Pure functions returning owned objects:** In Prusti pure function can currently only return values of types that implement the `Copy` trait, such as primitive types. This in particular excludes types that cause allocations, such as `String`. In real code bases such as OpenDP this complicates verification as many otherwise pure functions do return such owned objects. For example the `Error`s returned in OpenDP contain `String`s that provide diagnostic information, and as such these functions cannot be treated as pure in Prusti and consequently cannot be used in pre- and postconditions. Implementing support for this could benefit the verification effort on OpenDP.

V. Schedule

- Goal 1: 4 weeks for selection
- Goal 2: 12 weeks for implementation
- Goal 3: 2 weeks for evaluating results
- Extension Goals: 4 weeks
- Writing report: 4 weeks

For a total of 6 months.

References

[1] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: https://doi.org/10.1145/2663171.2663188

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360573

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[4] The OpenDP Team, "OpenDP Library." [Online]. Available: https://github.com/opendp/opendp

[5] M. Gaboardi, M. Hay, and S. Vadhan, "A programming framework for opendp," 2020. [Online]. Available: https://projects.iq.harvard.edu/files/opendifferentialprivacy/files/opendp_programming_framework_11may2020_1_01.pdf

[6] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, nov 2019. [Online]. Available: https://doi.org/10.1145/3324783

[7] 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*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–14. [Online]. Available: https://doi.org/10.1145/3519939.3523445

[8] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers, "Modular specification and verification of closures in rust," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485522