Specifying Smart Contract with Hax and ConCert

Lasse Letager Hansen
letager@cs.au.dk
Aarhus University
Denmark

Bas Spitters
spitters@cs.au.dk
Aarhus University
Denmark

Keywords: Coq, Hax, Hacspec, SSProve, ConCert, Smart Contract

1 Introduction

There has been a lot of progress in high-assurance cryptographic software. We aim to extend this to high-assurance smart contracts. One example of this effort is the Hax toolchain [7]. A part of this toolchain is the functional subset of Rust called Hacspec [4, 8], which aims to make cryptographic specifications executable and serve as reference implementations that can be used for testing more efficient implementations. We opt to use a functional smart contract language, as this facilitates proofs and translations, and there are already several functional smart contract languages [2]. Rust is a popular language for smart contracts, so we propose Hacspec as a sufficient language for smart contracts, as smart contracts face a lot of the same issues as high-assurance cryptographic software.

We could have extended other projects, such as Aeneas [6]; however, we believe the ideas and goals of Hacspec align better. Our goal is to write simple but detailed specifications of smart contracts for which we can prove relevant security properties. Allowing the use of borrows and stronger features will make specifications more complex and possibly introduce bugs. If we instead were to write an efficient implementation of a smart contract, Aeneas could be a useful tool or stepping stone. We therefore try to align with the community and Aeneas team about choices for the backends.

When formalizing, one either extracts or embeds. Smart contracts have historically used verified extraction. In this work, we extend a smart contract verification framework with an embedding. There already exist tools for reasoning about smart contracts, one of which is ConCert [3]; however, these are used by specialists and not the cryptographers and software engineers who write the smart contracts. Furthermore, the extracted code from ConCert is non-idiomatic Rust, which is non-trivial to make changes to. Going in the other direction, starting with Hacspec/Hax and producing Coq code, has the added benefit of producing clean Rust code. We can also state guarantees and properties that we want to hold directly in the code of the smart contract and then either manually or automatically prove these statements after translation to Coq.

2 Overview

We start by showing how to write a specification of a smart contract in Hacspec and how to add annotations to that specification. We then sketch how to translate the smart contract to Coq. Finally, we describe how the extra definitions are added based on the annotations to tell ConCert about the state, initialization, and entry points of the smart contract. See Figure 1.

We then discuss an example of a real-world smart contract and conclude on the impact of this framework.

3 Hax / Hacspec

Hacspec [4, 7, 8] is a high-assurance specification language. It is used by the Hax framework, which takes a program written in a subset of Rust and translates it into different backends (Coq, F*, EasyCrypt, etc.). The primary focus of this framework is to specify cryptographic primitives, but we have extended this translation to allow for the specification of smart contracts. There are three parts of a smart contract for which we add annotations to tell the backend how to define the smart contract, namely the data structures representing the state

```
#[hax::contract_state(contract = "name_of_contract")]
pub struct SomeContractState {
    ..
}
```

the initialization function, setting the initial state of the contract

```
#[hax::init(contract = "name_of_contract")]
pub fn init_some_contract(
    _: &impl HasInitContext
) -> InitResult<SomeContractState> {
    Ok(SomeContractState { .. })
}
```

and each of the functions represents an entry point into the smart contract

```
#[hax::receive(contract = "name_of_contract",
               name = "name_of_entry_point",
               parameter = "name_of_data_structure")]
pub fn some_function_name<A: HasActions>(
    ctx: &impl HasReceiveContext,
    state: SomeContractState,
) -> Result<(A, SomeContractState), ParseError> {
    let params: name_of_data_structure =
        ctx.parameter_cursor().get()?;
    ..
    Ok((A::accept(), state_ret))
}
```

The information we parse along to the backend for each entry point is the name of the contract, the name of the entry point, and the name of the data structure used as input for that entry point.

4 SSProve

We target the state-separating prove (SSProve) framework for Coq [1], which is one of the backends of Hax, as we also want to show security guarantees about the smart contracts. This allows us to define

CoqPL, Sat 20 January 2024, London, United Kingdom

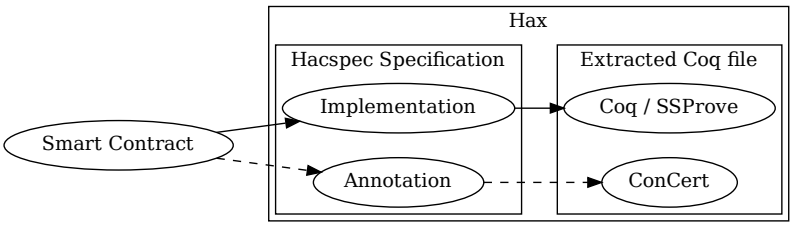

Figure 1. Translation of Smart Contract

security games, showing that an adversary can only distinguish the real protocol from an ideal oracle with negligible probability. Furthermore, the translation into SSProve comes with a proof of equivalence to the functional specification. This allows us to use the translation for computation, property-based testing, and much more.

5 ConCert

Now we have a way to specify and show security properties about individual functions; however, we also want to show safety properties about the smart contract as a whole. We do this by specifying the smart contract in ConCert [3], which is a framework for smart contract verification in Coq. This is done by using the annotations in the translation. The state is just renaming. The initial context is given by calling the function designated by the Rust annotation. Each of the entry points is translated into calls to their respective functions with arguments as given by the annotations. Now we define an inductive type for carrying the information about how to call the entry points

```
Inductive Msg : Type :=
| msg_name_of_entry_point : t_name_of_data_structure → Msg
| ...
```

and how to handle the calls

```
Equations receive chain ctx st msg
  : result (state * list ActionBody) t_ParseError :=
  receive_name_of_contract chain ctx st msg  :=
    match msg with
    | Some (msg_name_of_entry_point val) ⇒
      match (receive_name_of_entry_point val st) with
         | inl x ⇒ Ok ((fst x), [ ])
         | inr x ⇒ Err x
         end
    ...
    | _ ⇒
      Err tt
    end : result (state * list ActionBody) t_ParseError.
```

and finally, it is all packaged into a `Contract` type, which can be used by ConCert.

```
Definition contract : Contract state Msg state t_ParseError
     := build_contract init receive.
```

6 Example

For now, the framework is built to verify smart contracts from the Concordium blockchain [5]. We aim to combine the library of examples from Concordium and ConCert to get verified smart contracts specified in Hacspec, which can run on the Concordium blockchain. One such example is the wrapped currency contract for Concordium (wCCD), which has non-trivial guarantees we want to ensure. We can also specify and formalize part of the execution model for the blockchain by writing it in Hacspec and using the Hax toolchain.

7 Conclusion

Using an embedding for smart contracts allows software engineers to write clean code for real-world smart contracts. These can then be translated into Coq, where a verification framework can be used to verify correctness, safety, and cryptographic security guarantees. This toolchain is extendable enough to allow for writing guarantees and tests directly in the code for the smart contract, thus closing the gap on fully verified real-world smart contracts.

References

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. (2021). https://eprint.iacr.org/2021/397

[2] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming* 32 (2022), e11. https://doi.org/10.1017/S0956796822000077

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: a smart contract certification framework in Coq. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Jan 2020). https://doi.org/10.1145/3372885.3373829

[4] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. 2018. hacspec: Towards Verifiable Crypto Standards. In *Security Standardisation Research - 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*. 1–20. https://doi.org/10.1007/978-3-030-04762-7_1

[5] Ivan Damgård, Hans Gersbach, Ueli Maurer, Jesper Buus Nielsen, Claudio Orlandi, Torben P. Pedersen, Thomas Graham, Christian Matt, Mario Rogic, and Daniel Tschudi. 2020. Concordium White Paper [Vol. 1.0-April 2020]. https://api.semanticscholar.org/CorpusID:218626182

[6] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 711–741. https://doi.org/10.1145/3547647

[7] Franziskus Kiefer and Lucas Franceschino. 2023. Introducing hax. https://hacspec.org/blog/posts/hax-v0-1/.

[8] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. 2021. *hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.* Technical Report. Inria. https://hal.inria.fr/hal-03176482