Using Coq to Enforce the Checks-Effects-Interactions Pattern in DeepSEA Smart Contracts

Daniel Britten
The University of Waikato, Hamilton, New Zealand

Vilhelm Sjöberg
CertiK, New York, NY, USA

Steve Reeves
The University of Waikato, Hamilton, New Zealand

Abstract

Using the DeepSEA system for smart contract proofs, this paper investigates how to use the Coq theorem prover to enforce that smart contracts follow the *Checks-Effects-Interactions Pattern*. This pattern is widely understood to mitigate the risks associated with reentrancy. The infamous "The DAO" exploit is an example of the risks of not following the *Checks-Effects-Interactions Pattern*. It resulted in the loss of over 50 million USD and involved reentrancy – the exploit used would not have been possible if the *Checks-Effects-Interactions Pattern* had been followed.

Remix IDE, for example, already has a tool to check that the *Checks-Effects-Interactions Pattern* has been followed as part of the Solidity Static Analysis module which is available as a plugin. However, aside from simply replicating the Remix IDE feature, implementing a *Checks-Effects-Interactions Pattern* checker in the proof assistant Coq also allows us to use the proofs, which are generated in the process, in other proofs related to the smart contract.

As an example of this, we will demonstrate an idea for how the modelling of Ether transfer can be simplified by using automatically generated proofs of the property that each smart contract function will call the Ether transfer method at most once (excluding any calls related to invoking other smart contracts). This property is a consequence of following a strict version of the *Checks-Effects-Interactions Pattern* as given in this paper.

2012 ACM Subject Classification Security and privacy → Logic and verification; Computer systems organization → Distributed architectures

Keywords and phrases smart contracts, formal methods, blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2021.3

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021`; archived at `swh:1:dir:85ea91f0b51380b40bf760195c03a5564d195993`

Acknowledgements I (Daniel Britten) want to thank the University of Auckland and Associate Professor Jing Sun for kindly hosting me during this research.

1 Introduction

The importance of smart contracts being correct has been voiced many times, most obviously because of the high financial risk associated with a smart contract being incorrect and exploited (such as "The DAO" [7] and others [1, 6, 8]) which all involved the use of what we will refer to as *malicious reentrancy*.

Reentrancy involves a smart contract C that triggers the execution of code of another smart contract D which then calls a function in the original smart contract C before the original execution of C has completed. However, when not handled properly, reentrancy can

cause a smart contract to behave incorrectly and be exploited. This happens with malicious reentrancy, which maliciously exploits the situation that the original execution of C has not completed.

This issue can be mitigated by following the *Checks-Effects-Interactions Pattern* which suggests that a smart contract should first do the relevant *Checks*, then make the relevant internal changes to its state (*Effects*), and only then interact with other smart contracts which may well be malicious. When following the *Checks-Effects-Interactions Pattern* a reentrant call is essentially no different to a call that is initiated after the first call is finished so no additional risk from malicious reentrant calls is possible.

On the Ethereum blockchain, interacting with a malicious smart contract is even possible when transferring Ether. This is because if the recipient is a smart contract then it has the opportunity to run some code on receiving funds.

The problem with all this is that the modelling of smart contract execution when there is the possibility of reentrancy is difficult and the related correctness proofs would be complex as well. Even modelling the humble Ether transfer needs to take the possibility of reentrancy into account.

Using the DeepSEA [2] system for proofs about smart contract correctness, a method of enforcing the *Checks-Effects-Interactions Pattern* has been developed. Enforcing the *Checks-Effects-Interactions Pattern* greatly simplifies the modelling of any action that might involve external calls (including Ether transfers).

Tangibly, enforcing the *Checks-Effects-Interactions Pattern* means that the DeepSEA code for a smart contract function shown on the left (Listing 1) should not be permitted and the code shown on the right (Listing 2) should be allowed.

Listing 1 "Unsafe" function.

```
let unsafeExample() =
  transferEth(msg_sender, 0u42);
  transferSuccessful := true
```

Listing 2 "Safe" function.

```
let safeExample() =
  transferSuccessful := true;
  transferEth(msg_sender, 0u42)
```

The end result of the work in this paper is a system which automatically proves that the *Checks-Effects-Interactions Pattern* has been followed for most cases when it indeed has been, although there are some cases where the *Checks-Effects-Interactions Pattern* has been followed but this system cannot prove it, as a compromise for automation. A related result is then used to demonstrate an idea for simplifying the modelling of Ether transfers.

The main contributions of this paper are as follows:

- A Coq [3] proposition formalising the notion of a smart contract function following the *Checks-Effects-Interactions Pattern*. This is discussed in Subsection 2.4.
- Automated proofs related to the previous contribution as well as related automated proofs that the lists of transfers (that are directly generated by the smart contract) after function calls are of length at most one. See Section 3 and Section 4 respectively.
- A demonstration of an idea for simplifying the modelling of what states are reachable by a smart contract by making use of some of these automated proofs (Section 4).

2 Representing the absence of reentrancy situations as a proof goal

2.1 The DeepSEA system

All the modelling and proofs in the paper make use of the DeepSEA system for smart contract proofs. DeepSEA [2] is an up and coming framework and smart contract language that promises to provably link high-level specifications in Coq [3] to Ethereum Virtual Machine (EVM) bytecode. This will give a high degree of certainty that results proven about the high-level specifications also hold for the bytecode.

2.2 The *Checks-Effects-Interactions Pattern*

The *Checks-Effects-Interactions Pattern* suggests that a smart contract should follow a pattern in which calls to external contracts are always the last step [11].

When following the *CEIP*, nested calls are equivalent to calls invoked one after another as nested calls cannot influence the outcome of the original call (excluding considering gas). This enables a simpler model than completely modelling reentrancy with co-recursive functions. The simpler model is considered to be equivalent to a complete model in terms of modelling what states are reachable and we rely on an informal knowledge for this. Ideally, we would like to model reentrancy foundationally making use of the EVM semantics and then prove that the simple model is equivalent to the more complex model in the case where the *CEIP* is followed.

In this paper, a stricter version of the *Checks-Effects-Interactions Pattern* is used where only one *Interaction* is permitted. This eliminates modelling complications in the situations where two external calls are done but the first one turns out to throw an error. It is virtually impossible to know, when modelling, whether an arbitrary external call will throw an error, particularly due to the possibility of gas being exhausted.

This strict version of the *Checks-Effects-Interactions Pattern* will now simply be referred to as the *CEIP*.

2.3 Relevant aspects of the DeepSEA system

Listing 3 shows the same DeepSEA smart contract function in different representations. The intermediate level and high level representation are both generated automatically from the DeepSEA source. First, the intermediate level abstract syntax tree in Coq is generated from the source. The denotational semantics of the AST gives the high level representation (by the `synth_stmt_spec_opt` Coq function as a part of the DeepSEA system). The AST for each function contains the relevant information required to formulate the notion of whether the function adheres to the *CEIP*. The inductive proposition described in the next section makes use of the intermediate level AST representation.

2.4 Coq Inductive Proposition: `cmd_constr_CEI_pattern_prf`

The typing rule (Figure 1) corresponds to the definition of `cmd_constr_CEI_pattern_prf` which is an inductive proposition in Coq capturing the notion of a function following the *CEIP*. The typing rule is based upon the syntax of the smart contract as represented in the DeepSEA intermediate level language. It uses the assumption that reentrancy is only possible when certain syntax, such as `CCtransfer` is encountered. `CCtransfer` is the intermediate level language construct corresponding to a `transferEth` call. The 🟢 icon indicates that the contract cannot in any way have triggered reentrancy yet and the 🟠 icon indicates that

Listing 3 "Safe" function in different representations with similarities highlighted.

DeepSEA smart contract source code (not Coq):

```
let safeExample() =
  transferSuccessful := true ;
  transferEth(msg_sender, 0u42)
```

DeepSEA intermediate level language in Coq:

```
(CC sequence
(CCstore
  (LCvar Contract_transferSuccessful := true_var)
  (ECconst_int256 tint_bool true Int256.one))
(CC transfer
  (@ECbuiltin0 _ _ _ builtin0_caller_impl)
  (ECconst_int256 tint_U (Int256.repr 42_var))
  (Int256.repr 42))))
```

DeepSEA high level language in Coq:

```
(get;;
MonadState.modify (update_Contract_transferSuccessful true)) ;;
d <- get;;
(let (success, d') :=
me_transfer me (me_caller me) (Int256.repr 42) d in
if Int256.eq success Int256.one then put d' else mzero)
```

reentrancy may have been triggered by that point (and so no unsafe commands such as writing to storage should be allowed after that point). The 🔴 icon would indicate a contract that is vulnerable to malicious reentrancy but does not occur in the typing rule as the rule defines what is safe.

The transfer related rule in Coq is shown in Listing 4. The notion that at most one external call is allowed is captured by the fact that the proof requires the state `Safe_no_reentrancy` (🟢) beforehand. Due to the transfer the contract is then in a state where reentrancy may have occurred and this is captured by the state `Safe_with_potential_reentrancy` (🟠).

In Listing 5 we define that if the body of a for loop stays at state ρ (either 🟢 or 🟠) then the for loop as a whole is also defined to stay at state ρ.

The remaining definitions are available in the GitHub repository[1]. This defines what it means for a DeepSEA smart contract function to follow the *CEIP*. To be precise, if `cmd_constr_CEI_pattern_prf` can be proven for a given function then that function follows the *CEIP*.

A drawback of this formulation is that interrelated if statements are not able to be reasoned about. If the logical content of interrelated if statements made it possible to know the *CEIP* was indeed followed, this formulation would not allow those functions to be proved to be safe. This does however simplify proof automation. An alternative approach which made use of the high level representation of the smart contract was also explored. This "instrumented semantics" approach added reentrancy state tracking to the semantics of DeepSEA, and as a result *is* able to reason about interrelated if statements. This alternative approach still assumes specific syntactic elements correspond to the possibility of causing reentrancy.

[1] `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021` – See README for the specific files relevant to this paper.

Figure 1 Typing rule for a command that adheres to the *CEIP*, corresponding to the Coq inductive proposition `cmd_constr_CEI_pattern_prf`. (Some rarely used rules have been omitted). $\rho_x \in \{●, ●\}$.

$$\frac{}{\{\rho\}\ \texttt{skip}\ \{\rho\}} \qquad \frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\ \{\rho_3\}}{\{\rho_1\}\ \texttt{let}\ x = c_1\ \texttt{in}\ c_2\ \{\rho_3\}} \qquad \frac{}{\{●\}\ \texttt{load}\ \{●\}} \qquad \frac{}{\{●\}\ e_1 := e_2\ \{●\}}$$

$$\frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\{\rho_3\}}{\{\rho_1\}\ c_1\ ;\ c_2\ \{\rho_3\}} \qquad \frac{\{●\}\ c_{true}\{●\} \qquad \{●\}\ c_{false}\{●\}}{\{●\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{●\}}$$

$$\frac{\{●\}\ c_{true}\{●\} \qquad \{●\}\ c_{false}\{●\}}{\{●\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{●\}} \qquad \frac{\{●\}\ c_{true}\{●\} \qquad \{●\}\ c_{false}\{●\}}{\{●\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{●\}}$$

$$\frac{\{●\}\ c_{true}\{●\} \qquad \{●\}\ c_{false}\{●\}}{\{●\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{●\}} \qquad \frac{\{●\}\ c_{true}\{●\} \qquad \{●\}\ c_{false}\{●\}}{\{●\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{●\}}$$

$$\frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{\rho\}} \qquad \frac{\{\rho_1\}\ function\ \{\rho_2\}}{\{\rho_1\}\ function\ call\ \{\rho_2\}}$$

$$\frac{}{\{●\}\ \texttt{transferEth}(e_1, e_2)\ \{●\}} \qquad \frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{assert}\ c\ \{\rho\}} \qquad \frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{deny}\ c\ \{\rho\}}$$

Listing 4 Defining *CEIP* adherence for `CCTransfer`.

```
| CCCEIPtransfer :
 forall e1 e2,
  cmd_constr_CEI_pattern_prf
  _ (* Infer the return type *)
  Safe_no_reentrancy (* ● *)
  (CCtransfer e1 e2) (* Typically related to a 'transferEth' call. *)
  Safe_with_potential_reentrancy (* ● *)
    (* After, the possibility of reentrancy is noted. *)
```

Listing 5 Defining *CEIP* adherence for `CCFor`.

```
| CCCEIPfor :
 forall {ρ} id_it id_end e1 e2 c,
  cmd_constr_CEI_pattern_prf _ ρ c ρ
    (* Given a command that stays at state ρ *)
  -> cmd_constr_CEI_pattern_prf _ ρ (CCfor id_it id_end e1 e2 c) ρ
    (* Then the for loop as a whole stays at state ρ *)
```

Listing 6 Coq tactic to prove adherence to the *CEIP*.

```
Ltac CEI_auto :=
  repeat (
    reflexivity
  + typeclasses eauto
  + eapply CCCEIPskip + eapply CCCEIPlet + eapply CCCEIPload
  + eapply CCCEIPfor + eapply CCCEIPtransfer + ... ).
```

Another drawback (with both approaches) is that other techniques to manage reentrancy issues such as locks are not considered to be safe by these methods, even when they may have been used in a way which is safe. On the other hand, this does simplify modelling by only needing to consider cases equivalent to when no reentrancy occurs.

3 Automatically proving the absence of reentrancy situations

Now that we have defined the notion of a smart contract following the *CEIP* the goal is to automatically prove this for every function that does indeed follow the *CEIP* (or at least, most). The automation will be carried out by Coq tactics.

The tactic, partially shown in Listing 6, will repeatedly apply the constructors from the `cmd_constr_CEI_pattern_prf` definition along with resolving certain typeclass goals automatically. The + used to combine the tactics is critical to ensure the tactic backtracks as necessary because sometimes it is not the first matching constructor that is relevant.

See GitHub[2] for the full definitions of all the tactics involved. The proofs are done automatically and provide the user with an error if they fail (which would likely indicate the *CEIP* was not followed).

4 Simplifying the modelling of Ether transfer

The fact that we are following the *CEIP* simplifies the modelling of Ether transfer due to the fact that nested calls can be considered to be called one after another as no nested calls can influence the outcome of the original call (excluding gas considerations), as discussed in Section 2.2. This means that when considering what states are reachable it is sound to treat the transfer as only affecting Ether balances and ignore any other potential state changes. Also, since we are following the strict version of the *CEIP* we know that there is at most one call to `transferEth` which further simplifies the modelling.

When modelling Ether transfer in DeepSEA, at the end of a smart contract function call a list of transfers is produced and the modelled overall balances need to be updated based upon that list. If the list contains more than one element, how the balances should be updated is unclear due to the possibility of reentrancy having occurred. This is where a proof that only one transfer at most was directly generated is particularly useful. Coq allows us to pass this proof as an argument to our definition and use it to discharge the case where the list is longer than one element, as shown in Listing 7. This is greatly useful for simplifying the modelling by allowing us to demonstrate to Coq that we do not need to model reentrancy related to multiple transfers. If we did not have the proof we would be stuck

[2] `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021` – See README for the specific files relevant to this paper.

Listing 7 Updating balances for a list of length at most one.

```
Program Definition update_balances_from_transfer_list transfers
  ( length_evidence : length transfers <= 1 ) previous_balances a :=
match transfers with
| [] => previous_balances a
| [t] => update_balances_from_single_transfer contract_address
           (recipient t) (amount t) previous_balances a
| (h :: i :: t) as l => _ (* Coq allows us to discharge this case. *)
end.
Next Obligation. (* This is the case where transfers = h :: i :: t *)
intros.
exfalso. (* There is an impossible situation. *)
rewrite <- Heq_transfers in length_evidence. simpl in length_evidence. lia.
Defined.
```

with either truncating the list (which would be inaccurate) or assuming all the transfers took place with no reentrancy (which would also be inaccurate and leave the supposedly proven correct contract open to potential malicious reentrancy).

The relevant proofs that each smart contract function directly generates at most one transfer are similar to the proofs about the *CEIP* being followed in the sense that the DeepSEA `inv_runStateT_branching` tactic considers all branches of code execution like done by the `CEI_auto` tactic (Listing 6).

This technique simplifies the modelling of Ether transfer without leaving the door open for malicious reentrancy. The proofs are automated, only requiring the DeepSEA smart contract programmer to follow the strict version of the *CEIP*.

5 Related Work

A number of other tools aim to tackle the problem of reentrancy, such as [4, 5, 9] and [10]. This work is unique in that it explicitly makes use of proofs related to the *CEIP* in simplifying modelling smart contracts. It also is a step towards a smart contract proof system that uniquely targets the EVM as well as allowing proofs to be done on a high-level representation of the smart contract with strong guarantees that the properties proven about the high-level representation will also apply to the EVM bytecode.

6 Conclusion

This paper discusses an approach for representing and automatically proving that DeepSEA smart contracts follow the *CEIP* (code available on GitHub[3]). This is demonstrated by defining an inductive proposition in Coq that states that a particular smart contract function follows the *CEIP*. A proof that each smart contract function calls the Ether transfer function at most once is also discussed. An application of these proofs to simplify the modelling of Ether transfer is then discussed.

[3] `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021` – See README for the specific files relevant to this paper.

References

1 Lucas Campbell. DeFi platform dForce hacked for $25m - ERC777 reentrancy attack. `https://defirate.com/dforce-hack/`. (Accessed on 23 May 2021).

2 CertiK Foundation. DeepSEA. `https://github.com/certikfoundation/deepsea`. (Accessed on 23 May 2021).

3 The Coq Development Team. The Coq proof assistant. `https://coq.inria.fr/`. (Accessed on 23 May 2021).

4 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

5 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

6 Alex Manuskin. Living in a lego house: The imBTC DeFi hack explained. `https://www.zengo.com/imbtc-defi-hack-explained/`. (Accessed on 23 May 2021).

7 Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *Journal of Cases on Information Technology*, 21(1):19–32, 2019.

8 David Oz Kashi. The reentrancy strikes again – The case of Lendf.Me. `https://valid.network/post/the-reentrancy-strikes-again-the-case-of-lendf-me`. (Accessed on 23 May 2021).

9 Remix. Remix – Ethereum IDE. `https://remix.ethereum.org/`. (Accessed on 23 May 2021).

10 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

11 Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the Ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.