Gradual Verification for Smart Contracts

Haojia Sun
Shanghai Jiao Tong University

Kunal Singh
Carnegie Mellon University

Jan-Paul Ramos-Dávila
Cornell University

Jonathan Aldrich
Carnegie Mellon University

Jenna DiVincenzo
Purdue University

Abstract

Blockchains facilitate secure resource transactions through smart contracts, yet these digital agreements are prone to vulnerabilities, particularly when interacting with external contracts, leading to substantial monetary losses. Traditional verification techniques fall short in providing comprehensive security assurances, especially against re-entrancy attacks, due to the unavailable implementations of external contracts. This paper introduces an incremental approach: *gradual verification*. We combine static and dynamic verification techniques to enhance security, guarantee soundness and flexibility, and optimize resource usage in smart contract interactions. By implementing a prototype for gradually verifying Algorand smart contracts via the pyTEAL language, we demonstrate the effectiveness of our approach, contributing to the safe and efficient execution of smart contracts.

Keywords Logic, Gradual Verification, Smart Contracts

1 Introduction

Smart contracts, self-executing programs on blockchains like Algorand and Ethereum, facilitate secure resource transactions among mutually untrusted parties [4, 13]. Despite their potential, bugs in smart contracts come with serious consequences, such as substantial monetary loss [10]. Therefore, it is important to ensure the correctness of smart contracts.

Verification is particularly challenging when one smart contract calls another, which is external. The external contract may not be verified and may break the assumptions its caller makes, particularly in the case of re-entrant calls like those at the root of the DAO attack on Ethereum [9].

Recent verification approaches have been introduced to tackle this problem. Bräm et al. [2], Hajdu and Jovanović [5], Hildenbrandt et al. [6], and Kalra et al. [7] introduce static verification techniques capable of verifying the functional properties of Ethereum smart contracts, even when they involve calls to external contracts. These approaches provide strong guarantees, but they burden developers by demanding exhaustive and meticulous specifications. Rodler et al. [11], Shyamasundar [12], and Li et al. [8] introduce approaches that dynamically monitor and enforce user-specified invariants in smart contracts that call external contracts. While these dynamic techniques offer flexibility and ease of use, they can only uncover vulnerabilities in executed program paths at run time. Moreover, run-time checks incur substantial transaction fees on blockchains, increasing the monetary cost of executing smart contracts.

Given these constraints, gradual verification [1, 14] emerges as a compelling solution for securing smart contracts. Gradual verification supports partial specifications and incremental verification of code by applying static verification where possible and dynamic verification where necessary. This gives developers control over the trade-offs between static and dynamic verification. They can write more static specifications and get stronger guarantees and less run time overhead; or, they can write fewer specifications—saving on human effort —and rely more on run-time checking and its cost. The spectrum of trade-offs is formally guaranteed [1, 14]. We present a prototype for gradually verifying smart contracts via TEAL, the programming language used for creating Algorand smart contracts. By extending a gradual verification architecture from prior work [3, 14], we can provide the following benefits to smart contracts and their developers:

Protecting Against Unverified Code & Re-entrancy. Traditional static verification techniques are unsound in the face of unverified code and arbitrary re-entrancy. In contrast, gradual verification soundly guards against undefined behavior in unverified code and from arbitrary re-entrancy by runtime checking pre- and postconditions—that may be partially specified—on current and external contracts. Furthermore, for critical sections of available code, static verification can be employed proactively to identify potential issues.

Balancing Precision and Flexibility. The incremental approach to verification supported by gradual verification is well-suited for the fast-paced and ever-changing blockchain environment. It also makes verification more accessible to novice developers of smart contracts. Gradual verification suppresses static verification errors from missing specifications allowing developers to specify only the software components and properties they care about! Only true static and dynamic errors are reported, allowing bugs and vulnerabilities in smart contracts and their specifications to be detected earlier than static or dynamic verification alone [3].

Reducing Run-time Overhead. The transaction fees and computational resources required for executing smart contracts can accumulate quickly, like the Ethereum gas fee, which increases with dynamic verification alone. Gradual

PriSC'24, January 17–19, 2024, London, United Kingdom
2024.

verification minimizes this cost by minimizing run-time checks with statically available information—when a proof obligation has been statically verified, no run-time check is generated for it. Therefore, developers may choose to spend more time incrementally specifying their software to reduce dynamic checking overhead.

2 Architecture: Gradual pyTEAL

We present a pipeline (see Fig. 1) implementing the gradual verification of pyTEAL smart contracts, part of the Algorand platform. pyTEAL contracts are written in a domain-specific subset of Python, to which we added pre- and post-condition assertions. Our approach converts pyTEAL programs to Gradual C0, the only previously implemented gradual verifier [3]. Internally, Gradual C0 uses a variant of the Viper toolchain [2] to statically verify functions, treating partial specifications optimistically. The modified Viper back-end generates run-time checks wherever static verification could not assure specifications; these run-time checks are added to the original smart contract before compilation to the Algorand platform.

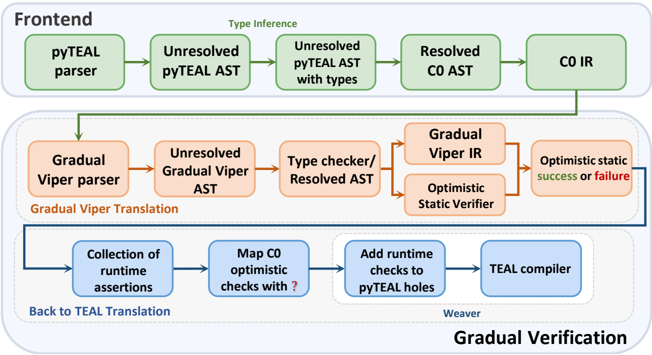
Frontend
Type Inference
pyTEAL parser
Unresolved pyTEAL AST
Unresolved pyTEAL AST with types
Resolved C0 AST
C0 IR
Gradual Viper parser
Unresolved Gradual Viper AST
Type checker/ Resolved AST
Gradual Viper IR
Optimistic Static Verifier
Optimistic static success or failure
Gradual Viper Translation
Collection of runtime assertions
Map C0 optimistic checks with ?
Add runtime checks to pyTEAL holes
TEAL compiler
Back to TEAL Translation
Weaver
Gradual Verification

Figure 1. Gradual pyTEAL verification pipeline

Front-end. We construct a parser using the FastParse Scala library to parse pyTEAL source code with specifications into an unresolved pyTEAL Abstract Syntax Tree (AST). Since Python supports dynamic typing but we are translating to a typed intermediate language (C0), we use type inference to assign static types to all variables in the pyTEAL AST, resulting in an unresolved pyTEAL AST with inferred types. We perform type checking and other well-formedness checks, then translate this "resolved AST" to C0. The C0 AST makes some operations more explicit, aiding verification and connecting to the intermediate representation supported by Gradual C0.

Static Verification. During static verification, the verifier *optimistically* interprets *imprecise formulas* to satisfy a contract (denoted by ?), with a promise that these holes in the specification will be dealt with at run time. When optimistic static verification succeeds, a set of run-time checks are produced that must be executed at run time for soundness.

Dynamic Verification. We then identify the locations where run-time checks should be inserted, based on the placement of ? embedded in our pyTEAL specifications. We developed the Weaver module to perform this step and encode the checks in the original pyTEAL program at the identified locations. Finally, the pyTEAL program with run-time checks is transformed into an executable contract with the pyTEAL compiler, which is subsequently executed on the Algorand platform. In the event of run-time check failures, corresponding error messages are reported. In practice, run-time checks may contain simple logical expressions, accessibility predicates denoting ownership of contract state, and complex predicates implemented as recursive boolean functions.

2.1 Example

```
#@ global Count;
@router.method
def sell(quantity: abi.Uint64):
  #@ requires ? and quantity>=0 and acc(Count);
  #@ ensures Count >=0;
  scratchCount = ScratchVar(TealType.uint64)
  return Seq(
    scratchCount.store(App.globalGet(
        Bytes("Count"))),
    App.globalPut(Bytes("Count"), scratchCount.
        load() - quantity.get())
  )
```

Figure 2. A gradually verified Algorand smart contract

In Figure 2, a segment of an Algorand smart contract depicts a selling transaction, where a specified quantity is subtracted from a global state variable Count. In Algorand smart contracts, global state refers to storage that persists across different contract calls and is accessible to all instances of the contract. The //@requires annotation in the code ensures that quantity is non-negative and verifies access to the Count variable. A ? indicates that this specification is partial. By specifying access permissions, the contract ensures that functions interact with the global state as intended, safeguarding against unauthorized modifications and security threats, thus maintaining the integrity of the state management.

Statically, the //@ensures annotation specifies a postcondition, asserting that Count remains non-negative after the transaction. This postcondition cannot be proven statically because an additional precondition, quantity <= Count, would be required. However, the verifier optimistically assumes this property can be derived from the ?, and a run-time check is added to make sure the property is obeyed at run time. By requiring the developer to explicitly specify data types, access controls, and preconditions/postconditions, the gradual verification architecture ensures that these specifications can be optimistically statically checked before the run time, while ensuring run-time soundness.

References

[1] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *Verification, Model Checking, and Abstract Interpretation: 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings 19*. Springer, 25–46.

[2] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J Summers. 2021. Rich specifications for Ethereum smart contract verification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[3] Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. *arXiv preprint arXiv:2210.02428* (2022).

[4] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer, 243–269.

[5] Ákos Hajdu and Dejan Jovanović. 2020. solc-verify: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*. Springer, 161–179.

[6] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

[7] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[8] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 438–453.

[9] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32.

[10] Malaw Ndiaye and Pr Karim Konate. 2021. Cryptocurrency crime: Behaviors of malicious smart contracts in blockchain. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE, 1–8.

[11] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[12] RK Shyamasundar. 2022. A Framework of Runtime Monitoring for Correct Execution of Smart Contracts. In *International Conference on Blockchain*. Springer, 92–116.

[13] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).

[14] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.