SMARTINV: Multimodal Learning for Smart Contract Invariant Inference

Presenter

Henry Blanchette

Smart Contracts

- *Definition.* A program deployed on a blockchain (e.g. Ethereum)
- Immutable once deployed, so, difficult to fix vulnerabilities post-hoc
- Highly desirable to prevent vulnerabilities *before* deployment
- Live vulnerability are *very expensive!!*
    - millions of USD in value are lost regularly

Smart Contracts – Functional and Implementation Bugs

Implementation Bugs

- Exhibit universal buggy behavior
- Example: integer overflow

Functional Bugs

- Newly-identified category of bugs
- “machine un-auditable”
- Cannot be reliably detected by relying on pre-defined bug patterns
  - Existing automated tools rely on such patterns

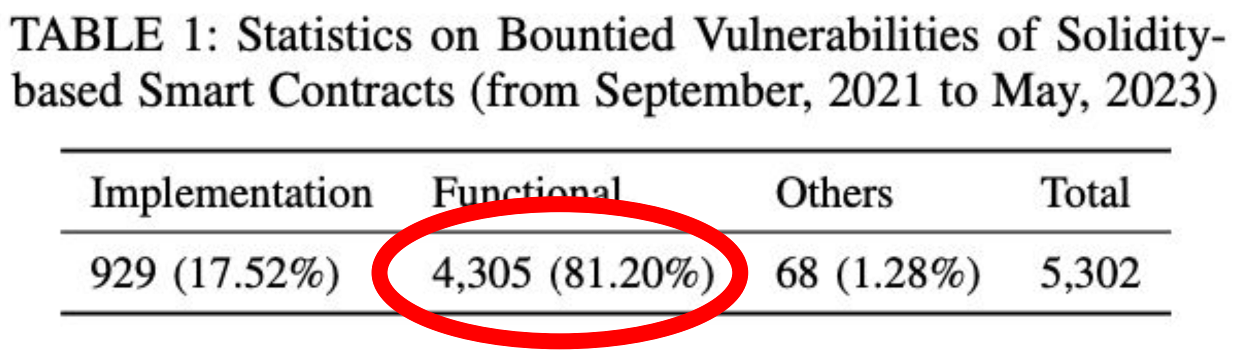


TABLE 1: Statistics on Bountied Vulnerabilities of Solidity-based Smart Contracts (from September, 2021 to May, 2023)

Implementation	Functional	Others	Total
929 (17.52%)	4,305 (81.20%)	68 (1.28%)	5,302

Smart Contracts – Functional Bugs as Difficult

Identification requires non-trivial reasoning across multiple multiple sources of information or modalities

- Example: reasoning across source code *and* natural-language documentation

SMARTINV

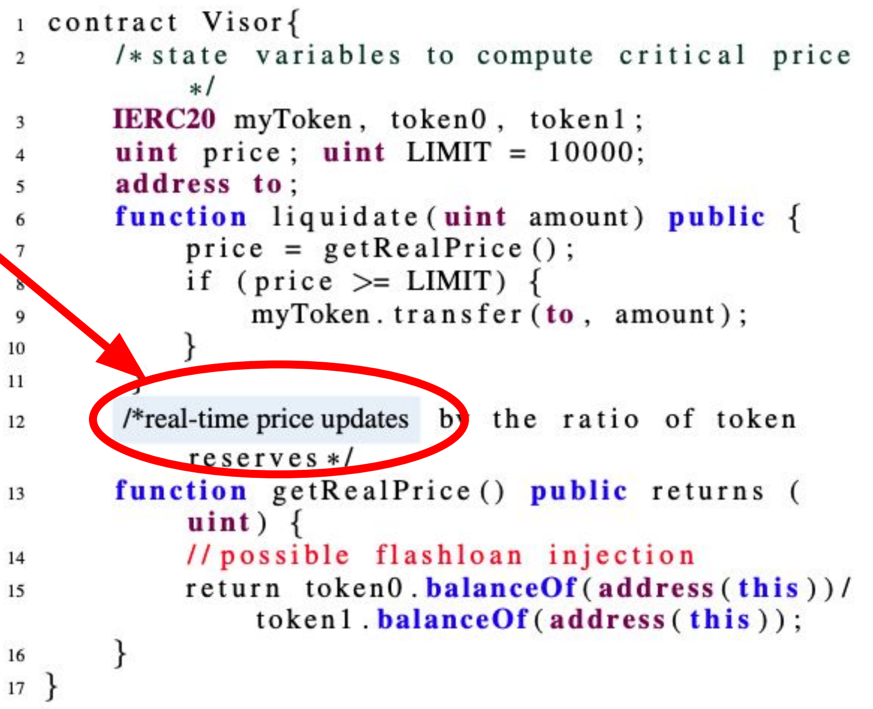
SMARTINV is a novel framework for:

- inferring smart contract invariants
- detecting bugs at scale

Example: Flashloan Primer

Implies real-time price oracle, which means price can change between beginning and end of flashloan

- A flash
  loan that allows users to borrow assets without cost as long as the loanee pays back within a *single transaction*
- Price manipulation hack:
  - Hacker injects a large flashloan into token0 price reserve, which changes price.
  - Hacker takes advantage of temporary price change to get a new flashloan at temporary better rate.
  - Hacker pays back loans, keeping an arbitrage profit.

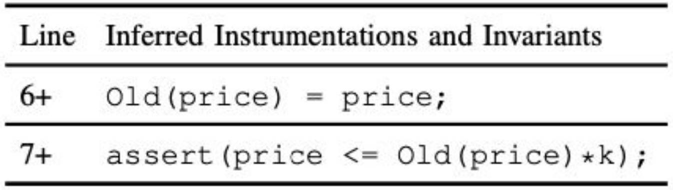


```
contract Visor{
    /* state variables to compute critical price
        */
    IERC20 myToken, token0, token1;
    uint price; uint LIMIT = 10000;
    address to;
    function liquidate(uint amount) public {
        price = getRealPrice();
        if (price >= LIMIT) {
            myToken.transfer(to, amount);
        }
    }
    /*real-time price updates by the ratio of token
        reserves*/
    function getRealPrice() public returns (
        uint) {
        //possible flashloan injection
        return token0.balanceOf(address(this))/
            token1.balanceOf(address(this));
    }
}
```

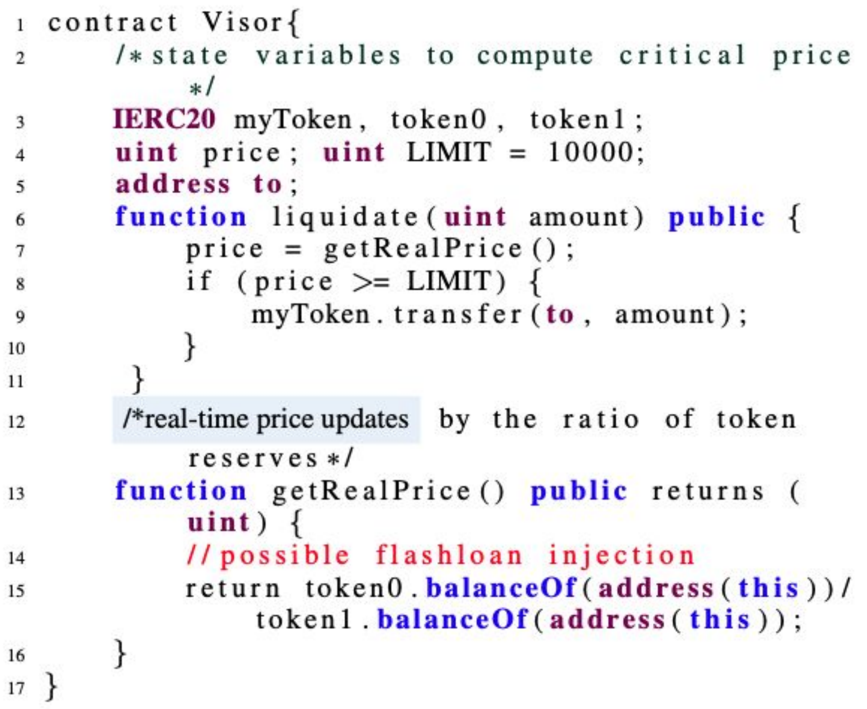
Listing 2: functional buggy snippet from $8-million-loss visor hack [73] (simplified for readability).

Example: Flashloan Primer

- Most traditional bug analyzers report this as a healthy contract
    - Based on formal verification, symbolic execution, dynamic analysis, common bug patterns
- Requires natural-language understanding of “real-time price updates” to recognize bug
- SMARTINV infers invariant by analyzing code source code and comment



Line	Inferred Instrumentations and Invariants
6+	`Old(price) = price;`
7+	`assert(price <= Old(price)*k);`



```
1  contract Visor{
2      /*state variables to compute critical price
           */
3      IERC20 myToken, token0, token1;
4      uint price; uint LIMIT = 10000;
5      address to;
6      function liquidate(uint amount) public {
7          price = getRealPrice();
8          if (price >= LIMIT) {
9              myToken.transfer(to, amount);
10         }
11     }
12     /*real-time price updates by the ratio of token
           reserves*/
13     function getRealPrice() public returns (
           uint) {
14         //possible flashloan injection
15         return token0.balanceOf(address(this))/
               token1.balanceOf(address(this));
16     }
17 }
```

Listing 2: functional buggy snippet from $8-million-loss visor hack [73] (simplified for readability).

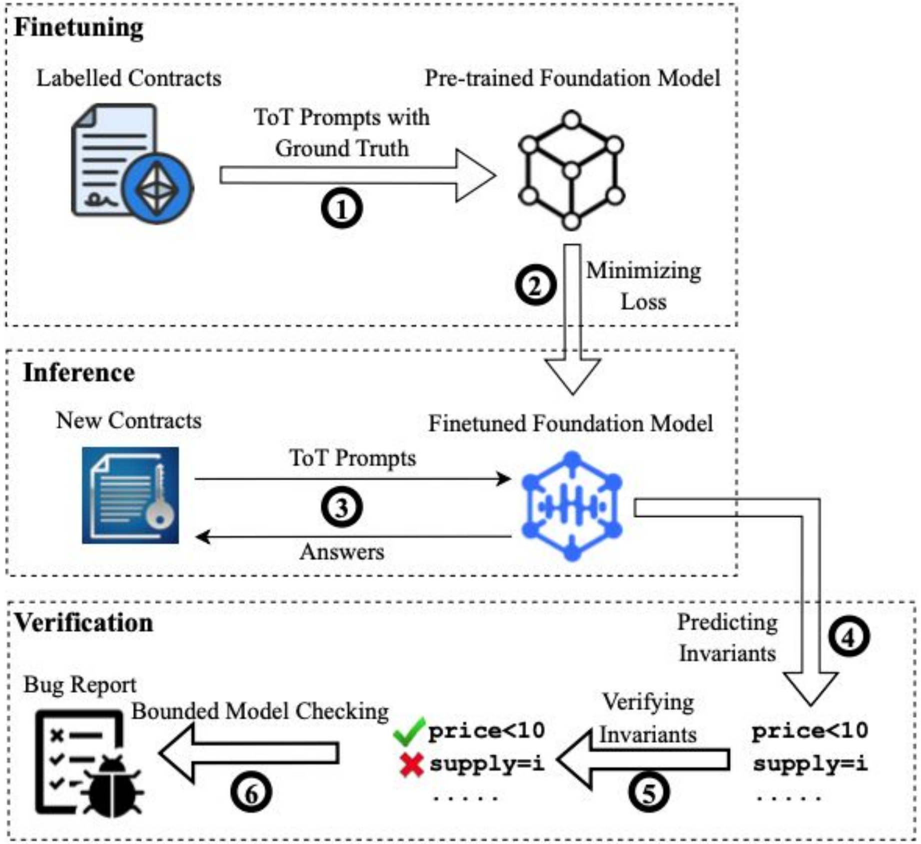
SMARTINV Workflow

Challenge #1: Incorporate and represent multimodal information *and* respective smart contract semantics

- Invariant templates and unique finetuning process that incorporates multimodal information

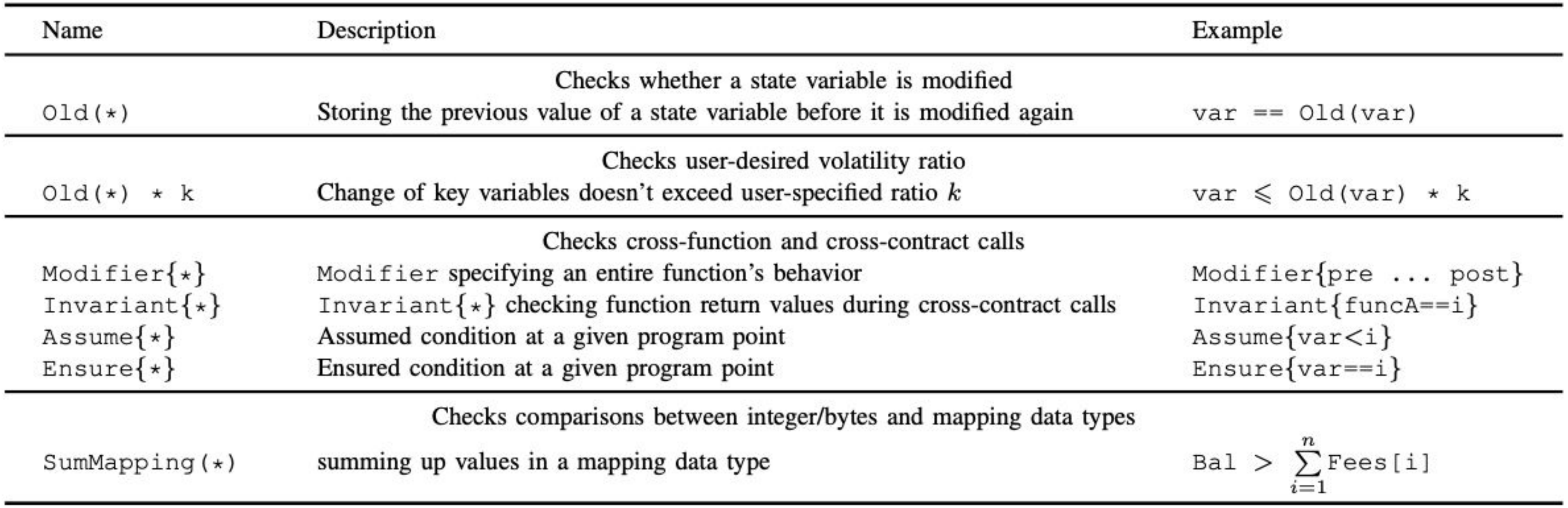
Challenge #2: Determine which invariants are correct (during inference)

- Novel invariant ranking strategy



SMARTINV – Invariant Templates

- Uses both standard and novel invariant templates
- Model is trained on smart contracts with labeled ground truth invariants
- Novel invariant templates:



Name	Description	Example
	Checks whether a state variable is modified	
`Old(*)`	Storing the previous value of a state variable before it is modified again	`var == Old(var)`
	Checks user-desired volatility ratio	
`Old(*) * k`	Change of key variables doesn't exceed user-specified ratio k	`var` $\leqslant$ `Old(var) * k`
	Checks cross-function and cross-contract calls	
`Modifier{*}`	`Modifier` specifying an entire function's behavior	`Modifier{pre ... post}`
`Invariant{*}`	`Invariant{*}` checking function return values during cross-contract calls	`Invariant{funcA==i}`
`Assume{*}`	Assumed condition at a given program point	`Assume{var<i}`
`Ensure{*}`	Ensured condition at a given program point	`Ensure{var==i}`
	Checks comparisons between integer/bytes and mapping data types	
`SumMapping(*)`	summing up values in a mapping data type	`Bal` $> \sum_{i=1}^{n}$ `Fees[i]`

SMARTINV – Tier of Thought Finetuning and Inference

- *Goal.* Guide a pre-trained foundation model towards generating gun-preventive invariants
- *New Idea.* Introduce increasingly complex thoughts into the conversation as the model is reasoning
  - (Model is finetuned to generate one thought at a time)
  - Tier 1: identify critical program points
  - Tier 2: identify possible invariants
  - Tier 3: rank possible invariants and bugs
- *Inference.* decompose the invariant generation problem into this 3-tiered task

SMARTINV – ToT Invariants Verification Algorithm

- As inference is generating possible invariants, convert the generated Solidity code to Boogie, which performs a static analysis to check if there is enough information present in the source code to prove program correctness
- If the invariant is strong enough, its marked as so
- If the invariant is *not* strong enough, model checker will search for counterexamples to propose to a Tier 3 reasoning prompt

Implementation

- Training and inference are written in 4011 LoC Python
- Invariant verification algorithm are written in 1322 LoC C#, using VeriSol
- Invariant templates are written in 169 LoC Solidity
- LLaMA-7B with 8-bit quantization used as foundation model
- LLaMA-7B was finetuned with Parameter Efficient Finetuning and low-rank adaptation
- Smart contract dataset of 179,319 contracts from 1 Jan 2016 to 1 July 2023 collected from Etherscan and public Github repositories using Google BigQuery
- Training contracts were labeled with ground truth features: transactional contexts, critical program points, all relevant invariants, critical invariants, ranked critical invariants, and vulnerabilties

Evaluation

Research questions:

- RQ1: In terms of bug detection, how does SMARTINV compare to six state-of-the-art bug analyzers and three similar prompting-based tools?
- RQ2: In terms of invariants generation, how does SMARTINV compare to similar tools?
- RQ3: How much do our selected model LLaMA and optimizing strategies improve the accuracy of bugs detection and invariants generation?
- RQ4: How fast is SMARTINV compared to similar tools?

RQ1: Effectiveness of Predicted Invariants for Bug Detection

Compared SMARTINV to several state-of-the-art bug-detection systems.

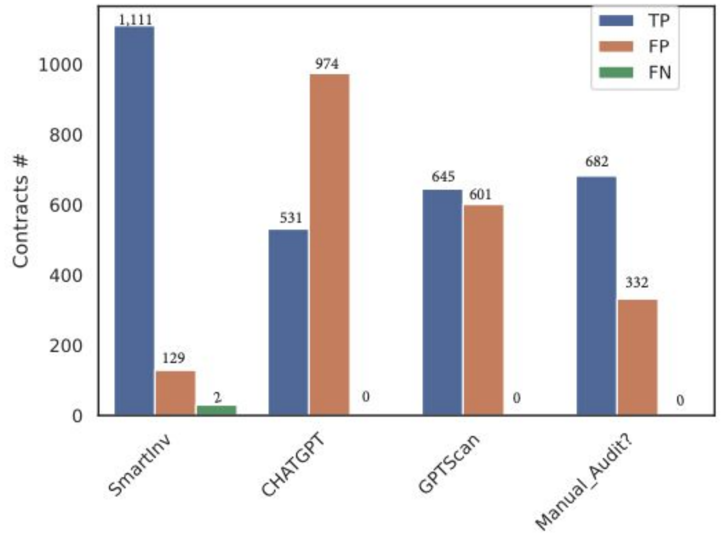
TABLE 7: Reported bugs breakdown by type from 89,621 contracts. The last seven rows capture a tool's aiblity to detect functional bugs.

Bug Type	SMARTINV	VERISOL	SMARTEST	VERISMART	MYTHRIL	SLITHER	MANTICORE
RE	9,011	1,591	0	0	1,311	2,533	901
IF	13,531	2,031	31655	29,015	602	952	421
AF	11,009	905	10,921	12548	648	0	421
SC	908	0	452	366	99	972	122
EL	611	0	0	0	82	1,200	34
IG	494	0	0	2	12	78	122
IVO	1,022	4899	3,091	3,001	23	79	90
PM	2,651	5	2	0	0	0	0
PE	3,019	0	0	0	0	0	0
BLF	1,091	84	0	0	0	0	0
IS	977	33	5	5	107	0	0
AV	2,065	0	0	0	0	0	0
CB	3,192	0	0	0	0	0	0
IDV	1,924	0	0	0	0	0	0
Total Bugs	51,505	1,924	46,126	44,937	2,884	5,814	2,111

TABLE 8: Report on total count of error and timeout results from 89,621 contracts.

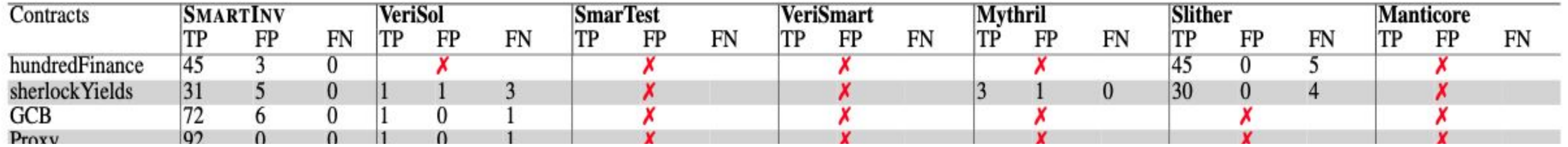
	SMARTINV	VERISOL	SMARTEST	VERISMART	MYTHRIL	SLITHER	MANTICORE
Error	0	87,697	11,859	11,859	14,211	83,807	23,301
Timeout	0	0	31,636	32,825	72,526	0	64,209

Figure 2: Comparison of SMARTINV with three prompting-based tools using the 1,241 contracts in the refined experiments of Table 9.



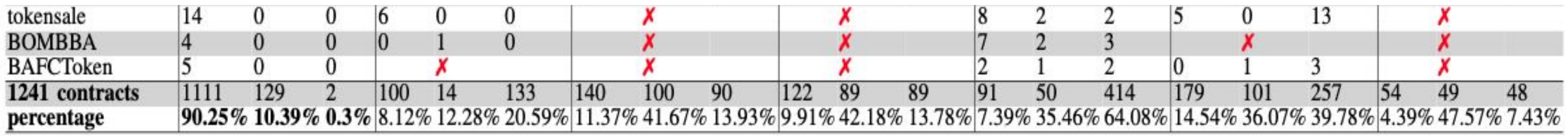
RQ1: Effectiveness of Predicted Invariants for Bug Detection

TABLE 9: Refined bug detection analysis on sampled 1,241 real-world buggy contracts with report on correct (TP), incorrect (FP), and missed (FN) bug alarms. ✗: a tool did not produce any results.



Contracts	SmartInv			VeriSol			SmarTest			VeriSmart			Mythril			Slither			Manticore		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
hundredFinance	45	3	0		✗			✗			✗			✗		45	0	5		✗	
sherlockYields	31	5	0	1	1	3		✗			✗		3	1	0	30	0	4		✗	
GCB	72	6	0	1	0	1		✗			✗			✗			✗			✗	
Proxy	92	0	0	1	0	1		✗			✗			✗			✗			✗	

⋮

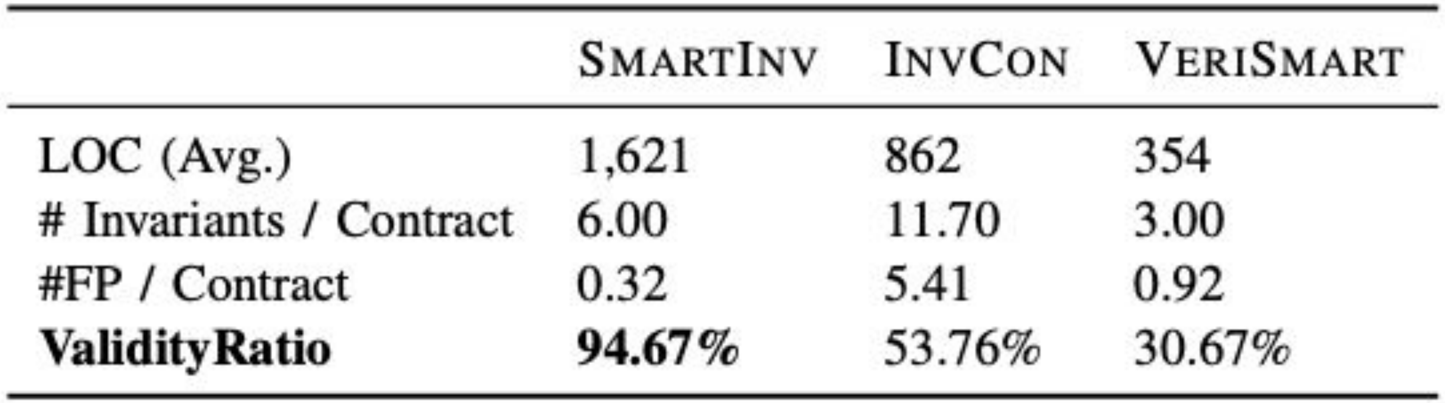


tokensale	14	0	0	6	0	0		✗			✗		8	2	2	5	0	13		✗	
BOMBBA	4	0	0	0	1	0		✗			✗		7	2	3		✗			✗	
BAFCToken	5	0	0		✗			✗			✗		2	1	2	0	1	3		✗	
1241 contracts	1111	129	2	100	14	133	140	100	90	122	89	89	91	50	414	179	101	257	54	49	48
percentage	**90.25%**	**10.39%**	**0.3%**	8.12%	12.28%	20.59%	11.37%	41.67%	13.93%	9.91%	42.18%	13.78%	7.39%	35.46%	64.08%	14.54%	36.07%	39.78%	4.39%	47.57%	7.43%

RQ2: Invariants Detection Accuracy

Compared SMARTINV's invariants detection capability with:

- INVCON (a Daikon-adapted smart contract invariants detector)
- VERISMART (a CEGIS-style verifier)



	SMARTINV	INVCON	VERISMART
LOC (Avg.)	1,621	862	354
# Invariants / Contract	6.00	11.70	3.00
#FP / Contract	0.32	5.41	0.92
ValidityRatio	**94.67%**	53.76%	30.67%

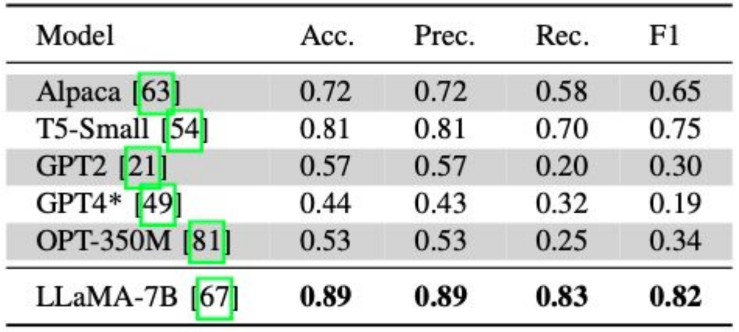
RQ3: Ablation Study

Considered six foundation models as our baselines: OPT-350M, Google’s T5-Small, OpenAI’s GPT2 and GPT4 (no finetuning available), Stanford’s Alpaca, and Meta’s LLaMA-7B

Goal. Quantify how much our key optimization strategies improved the end results

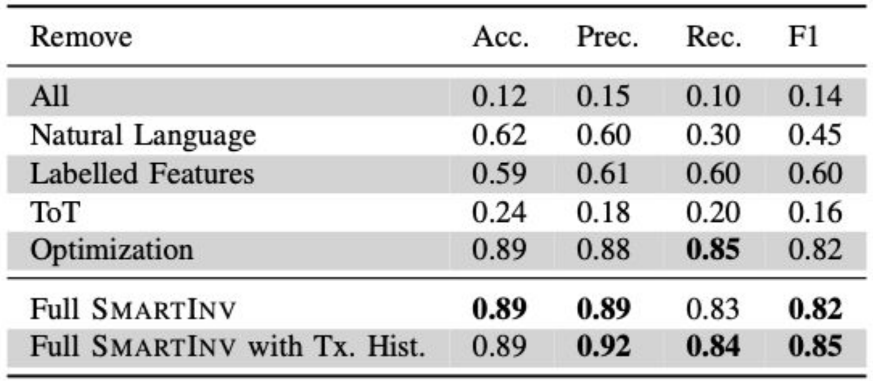
Method. Removed natural language modality in dataset by deleting comments and renaming functions to not give away domain-specific info.

TABLE 11: Finetuned candidate model eval. (*GPT4 results were obtained from prompt engneering alone without fine-tuning on curated training dataset due to close source).



Model	Acc.	Prec.	Rec.	F1
Alpaca [63]	0.72	0.72	0.58	0.65
T5-Small [54]	0.81	0.81	0.70	0.75
GPT2 [21]	0.57	0.57	0.20	0.30
GPT4* [49]	0.44	0.43	0.32	0.19
OPT-350M [81]	0.53	0.53	0.25	0.34
LLaMA-7B [67]	**0.89**	**0.89**	**0.83**	**0.82**

TABLE 12. Ablation Study



Remove	Acc.	Prec.	Rec.	F1
All	0.12	0.15	0.10	0.14
Natural Language	0.62	0.60	0.30	0.45
Labelled Features	0.59	0.61	0.60	0.60
ToT	0.24	0.18	0.20	0.16
Optimization	0.89	0.88	**0.85**	0.82
Full SMARTINV	**0.89**	**0.89**	0.83	**0.82**
Full SMARTINV with Tx. Hist.	0.89	**0.92**	**0.84**	**0.85**

RQ4: Runtime Performance

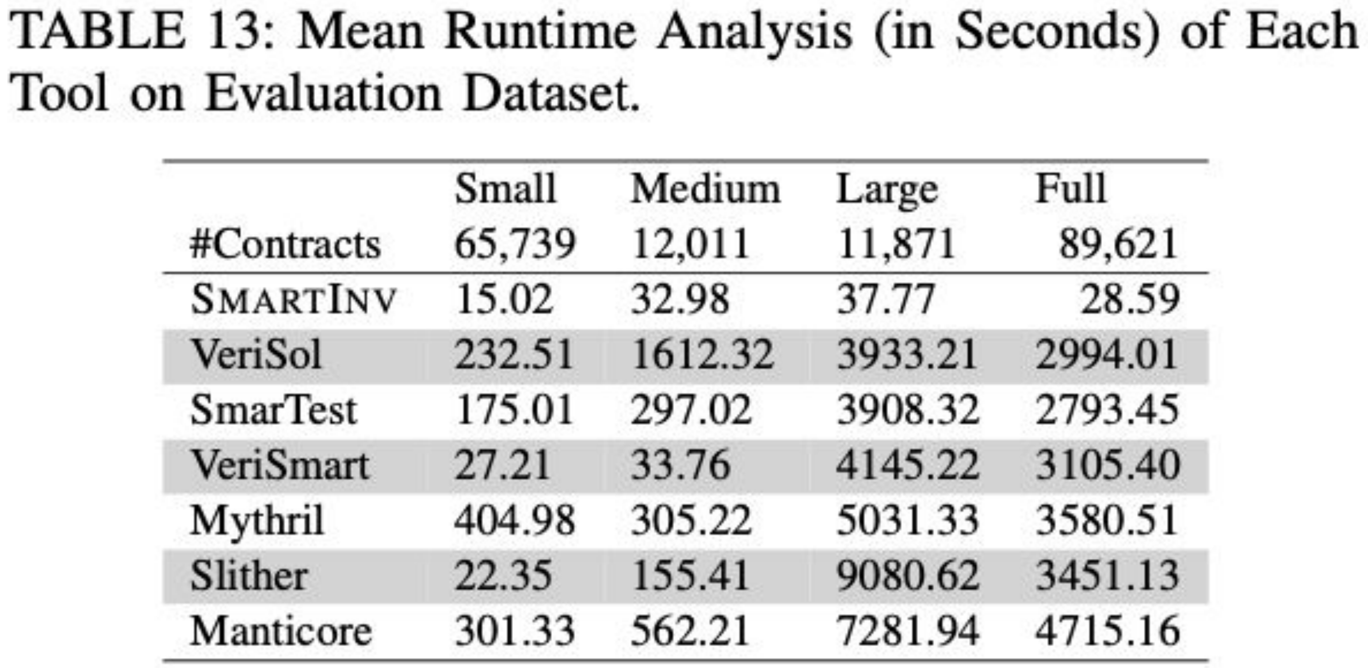


TABLE 13: Mean Runtime Analysis (in Seconds) of Each Tool on Evaluation Dataset.

	Small	Medium	Large	Full
#Contracts	65,739	12,011	11,871	89,621
SmartInv	15.02	32.98	37.77	28.59
VeriSol	232.51	1612.32	3933.21	2994.01
SmarTest	175.01	297.02	3908.32	2793.45
VeriSmart	27.21	33.76	4145.22	3105.40
Mythril	404.98	305.22	5031.33	3580.51
Slither	22.35	155.41	9080.62	3451.13
Manticore	301.33	562.21	7281.94	4715.16

Discussions

- *Token Length.* Foundation models have limited token length available for finetuning. LLaMA models are limited to 4069 tokens. So, large contracts were truncated. Future work to do here.
- *Verifier Compatibility with Solidity Compilers.* Based verifier on VeriSol's mapping between Solidity and Boogie, but this limits compiler versions and thus compatible contracts.
- *Exploitability of Zero-Day Bugs.* Not *all* detected zero-day bugs are exploitable, due to old compiler versions used in study.

Related Work

Smart Contract Static and Dynamic Analysis

- Symbolic execution, relying on common bug patterns
  - Can’t generalize beyond pre-determined bug patterns
- Static analysis tools use data flow analysis
  - Can’t be multimodal
- Fuzzing using randomized testing
  - Slow and unpredictable

Invariants Detectors and Verifiers

- Daikon and InvCon automatically generate possible invariants to consider
- SMTChecker, SolcVerifier, VeriSol, and Zeus require manual specification of invariants then infer transaction invariants
  - Requires good specification

ML-Based Tools

- SVChecker and Neural Contract Graph use NNs to discover sets limited sets of implementation bugs based on trained patterns
- Prompting-based tools don’t check for model hallucinations
- Overall, can’t reliably detect critical *functional* bugs

Scientific Peer Reviewer

Pranav Sivaraman

Technical Correctness

1. No Apparent Flaws

- The authors go into detail about their methodology. They explain their new prompting strategy and how it results in better model performance after fine-tuning, followed by formal verification.
- Comparison of SmartInv with existing tools, with SmartInv achieving the lowest False Positive Rate and False Negative Rate.
- Concludes with an ablation study of their framework and discusses the limitations of their framework, which are acceptable.

Scientific Contribution

2, 3, 4, 5, 6

- The authors present a new prompting technique and dataset that result in a tool designed to easily identify and fix known types of security vulnerabilities.
- The findings from this work could inspire future research that improves bug detection, enables more fixes in complex scenarios, and reduces runtime costs.

Presentation and Recommended Decision

3. Major but Fixable Flaws in Presentation

- The language of the paper can be adjusted to make it a bit more clearer. In particular a more detailed background on what smart contracts are would be helpful.
- Another issue is that the authors mention smart contracts are immutable, but don't explain how the bugs of a smart contract could be fixed.
- Table 8 should be replaced with a graph that better illustrates the author's claims.

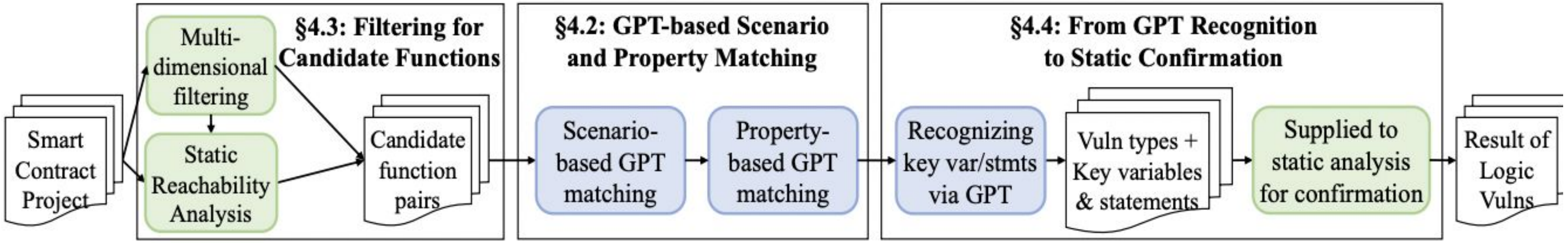
2. Accept with Noteworthy Concerns in Meta Review

Archaeologist

Aditya Ranjan

Previous Work

- GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis
- an early attempt to leverage LLMs for smart contract vulnerability detection
- combined GPT with traditional program analysis techniques to identify potential vulnerabilities
- breaks down vulnerability types into scenarios and properties
    - matches these to the candidate functions and uses static analysis tools with the AST to confirm that the code is vulnerable


Smart Contract Project
Multi-dimensional filtering
Static Reachability Analysis
§4.3: Filtering for Candidate Functions
Candidate function pairs
§4.2: GPT-based Scenario and Property Matching
Scenario-based GPT matching
Property-based GPT matching
§4.4: From GPT Recognition to Static Confirmation
Recognizing key var/stmts via GPT
Vuln types + Key variables & statements
Supplied to static analysis for confirmation
Result of Logic Vulns

Comparison

- lacks the ability to reason about multimodal input (natural language and source code)
- doesn’t have the ability to generate invariants to assist with preventing logical bugs in the contract
- doesn’t structure the prompting intelligently like SMARTInv does with increasing difficulty as you progress through the tiers
- SMARTINV’s fine-tuning approach helps overcome the limitations of pre-training knowledge that GPTScan mentions

Subsequent Work

- PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation
- uses a vector database of existing properties for retrieval-augmented generation of new properties
- more sophisticated process for ensuring the quality of generated properties
    - compilability checks, appropriateness weighted ranking, and runtime verification
- also defines its own intermediate language for property specification

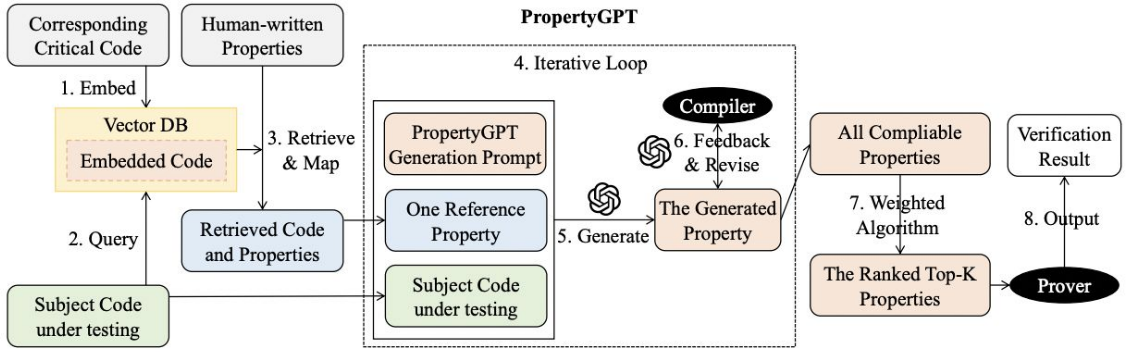


Figure 1: A high-level workflow of PropertyGPT.

Comparison

- retrieval mechanism using a vector database of existing properties rather than fine-tuning.
- iterative refinement process guided by compiler feedback and more rigorous checks
- more complicated ranking of invariants using similarity-based weight ranking algorithm
- SMARTINV’s multimodal reasoning capabilities are still unique strengths not explicitly addressed in PropertyGPT

Industry Practitioner

Sriman Selvakumaran

Smart contract bugs are **costly!**

$45 million lost over only 16 bugs in FY2024 Q1 (hacken.io)

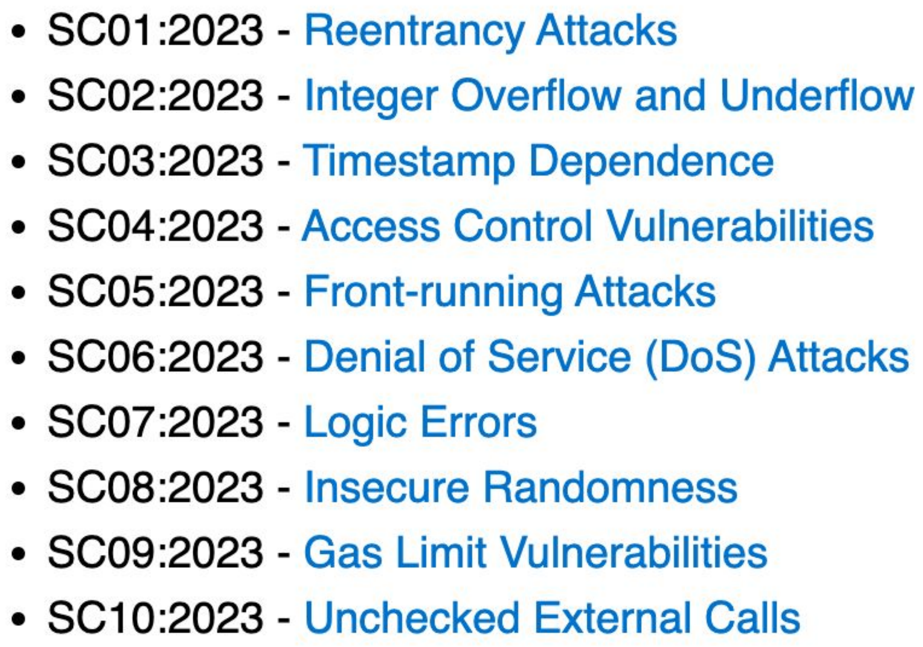
DeFi platform Compound falsely awarded **$90 million** due to vulnerabilities

August 2021 - hacker exploited bug to steal **$600 million** in tokens (ICBA)

Extremely diverse range of bugs

Many of the top smart contract bugs are implicit (hard-to-catch with static analyzers) or easily obfuscated

Smart contracts (like the blockchain) is **immutable** - cannot be modified in post



- SC01:2023 - Reentrancy Attacks
- SC02:2023 - Integer Overflow and Underflow
- SC03:2023 - Timestamp Dependence
- SC04:2023 - Access Control Vulnerabilities
- SC05:2023 - Front-running Attacks
- SC06:2023 - Denial of Service (DoS) Attacks
- SC07:2023 - Logic Errors
- SC08:2023 - Insecure Randomness
- SC09:2023 - Gas Limit Vulnerabilities
- SC10:2023 - Unchecked External Calls

*OWASP Smart Contract Top 10

Upshot: Definitely worth it

Proposed solution works better than static analyzers (according to their own metrics)

Can be used in tandem with static analyzers

Code is open-source

Little to no upfront cost to run

Code can be verified / modified

Risk is too high to not rigorously test

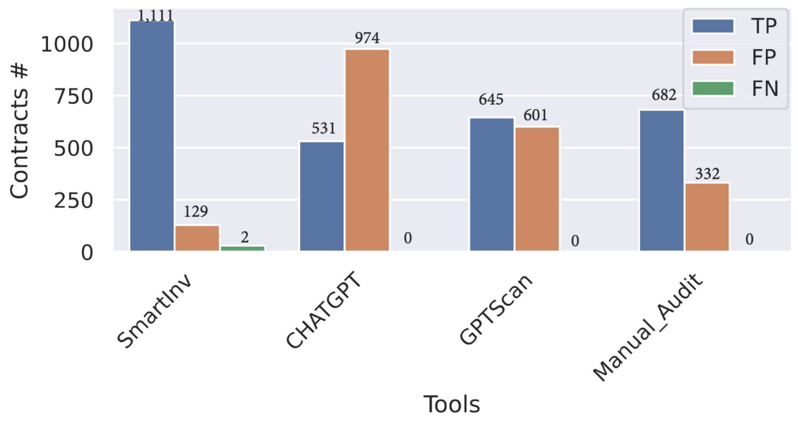


Figure 2: Comparison of SMARTINV with three prompting-based tools using the 1,241 contracts in the refined experiments of Table 8.

Con: Solution is still not perfect

Paper's solution works well up to a point

- ~10.5% of contracts false positives

Partially depends on in-file comments

- If no comments, can be ineffective

Must be used with other alternatives for maximum effectiveness

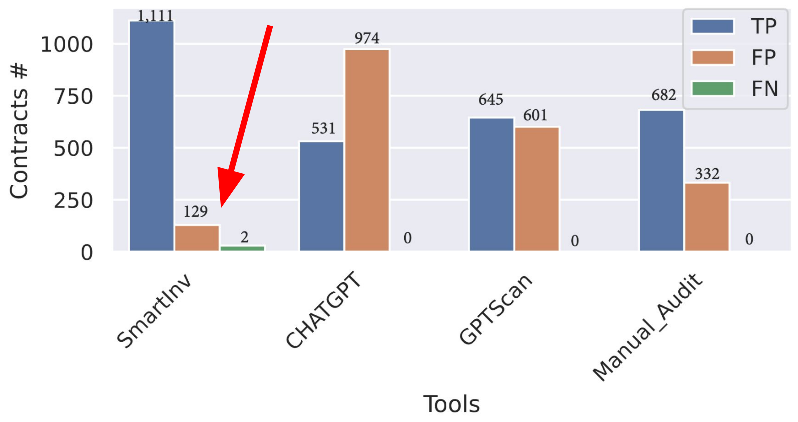


Figure 2: Comparison of SMARTINV with three prompting-based tools using the 1,241 contracts in the refined experiments of Table 8.

Private Investigator

Pranav Dulepet

Sally Junsong Wang

- Background
  - Researcher with affiliations to Stanford University, Columbia University, and the University of Cambridge
  - Research interests span security, systems, and law
  - Published works on smart contract security and financial systems accountability
- Relation to the paper
  - The paper aligns with her research focus on security and systems
  - Demonstrates her expertise in applying advanced computational techniques to blockchain and smart contract security
  - Combines her technical knowledge with insights into legal and financial systems, which is relevant for smart contract applications

Kexin Pei

- Background
    - Assistant Professor in the Department of Computer Science at the University of Chicago since July 2023
    - PhD in Computer Science from Columbia University (2016-2023)
    - MS in Computer Science from Purdue University (2014-2016)
    - BA in Computer Science from Hong Kong Baptist University (2010-2014)
    - Industry experience through internships at Google DeepMind (2022-2023) and Microsoft Research (2018-2019)
    - Research focuses on security, software engineering, and machine learning
    - Specializes in data-driven program analysis to improve security and reliability of software systems
- Relation to the paper
    - The paper directly applies his expertise in machine learning for program analysis
    - Demonstrates his focus on developing models that reason about program structure and behavior
    - Aligns with his goal of efficiently analyzing, detecting, and fixing software vulnerabilities
    - Showcases his ability to apply advanced techniques to real-world security problems in smart contracts

Junfeng Yang

- Background
  - Professor at Columbia University
  - PhD advisor to Kexin Pei from 2016 onwards
  - Research centers on making reliable and secure systems
  - Focus areas include
    - Security and robustness of machine learning
    - Tools to protect, verify, analyze, test, and debug software
    - Programming and runtime systems for cloud applications
  - BS in Computer Science from Tsinghua University
  - PhD in Computer Science from Stanford University
  - Created eXplode, a system for finding storage system errors (OSDI best paper award)
  - Worked at Microsoft Research Silicon Valley in 2008
- Relation to the paper
  - The paper aligns with his research focus on software security and reliability
  - Demonstrates his expertise in applying machine learning to improve system security
  - Builds on his work in developing tools for software verification and analysis
  - Showcases his ongoing collaboration with former students (Kexin Pei) in cutting-edge research
  - Applies his knowledge of distributed systems and cloud applications to the domain of smart contracts

Academic Researcher

Divahar Sivanesan

Project: SMARTFIX—Automated Repair of Smart Contracts Using Foundation Models

- SMARTFIX: an automated framework for not only detecting bugs and generating invariants in smart contracts but also automatically repairing the identified vulnerabilities.
- Introduce automated code repair mechanisms guided by the inferred invariants and leveraging the ToT prompting strategy.
- How? Extend the capabilities of foundation models to not only detect bugs and generate invariants but also to suggest code fixes that adhere to the inferred invariants and preserve the intended functionality of the contracts.

Core components and approaches

Proposed Flow

- SMARTINV: Input → ToT Tiers 1-3 → Verification → Bug Detection Output.
- SMARTFIX: Input → ToT Tiers 1-4 → Code Repair Module → Re-Verification → Fixed Code Output → Developer Interaction.
- Reminder:
  - Tier 1: Identifies critical program points where bugs are likely to occur.
  - Tier 2: Generates relevant invariants associated with these critical points.
  - Tier 3: Ranks the critical invariants to prioritize which ones are most crucial for the contract's correctness.
- Tier 4 Process:
  - Input: The smart contract code, the identified bugs, and the critical invariants that are currently violated.
  - Action: The model uses this information to generate specific code changes that would fix the bugs.

Automated Code Repair Module

- Data preparation:
  - Collect Bug-Fix Pairs: Assemble a dataset of buggy smart contracts and their corresponding fixed versions, including annotations explaining the bugs and fixes.
  - Fine-Tune the Model: Train the foundation model on this dataset to learn patterns of common bugs and their repairs.
- Integration with ToT Strategy:
  - Extend ToT with Repair Tier:
    - Tier 4: Generate code modifications to fix the detected bugs while ensuring that the critical invariants hold.
  - Example Prompt:
    - “Given the smart contract code and the following violated invariants, suggest code modifications to fix the contract while ensuring the invariants are satisfied.”

Verification of Generated Fixes

- Re-Verification Pipeline:
    - Syntax and Semantic Checks: Verify that the modified code compiles and adheres to Solidity's syntax and semantics.
    - Inductive Verification and BMC: Use the existing verification methods to ensure that the fixes satisfy all critical invariants and do not introduce new bugs.
    - Automated Testing:
        - Generate test cases based on the invariants to validate the functionality of the fixed contract.

Developer Interaction Interface

- User Interface:
    - Display Bugs and Fixes: Provide a platform where developers can see detected bugs and the model's suggested fixes side by side.
    - Interactive Editing:
        - Allow developers to accept, reject, or modify the suggested fixes directly within the interface.
- Feedback Mechanism:
    - Collect Developer Input: Gather feedback on the usefulness and accuracy of the fixes.
    - Iterative Improvement: Use this feedback to refine the model through additional fine tuning.

Challenges

- Maintaining Functional Correctness:
    - Challenge: Ensuring that code fixes do not alter the intended behavior of the contract.
    - Approach: Verify fixes using the existing verification pipeline.
- Model Generalization:
    - Challenge: The model may struggle to generalize fixes to unseen bugs or novel contract patterns.
    - Approach: Expand and diversify the training dataset with various bug types and contract styles. Employ regularization techniques to prevent overfitting.

Impact

- Enhanced Security:
  - Quick Mitigation: Reduces the time between bug detection and deployment of fixes, minimizing the risk of exploitation.
  - Preventative Measures: Helps in proactively securing contracts before deployment.
- Increased Developer Productivity:
  - Automation of Routine Tasks: Frees developers from manual debugging and patching, allowing them to focus on innovation.
  - Assistance for Less Experienced Developers: Provides guidance in secure coding practices, potentially raising the overall quality of smart contract code.