SmartPulse: Automated Checking of Temporal Properties in Smart Contracts

Jon Stephens*, Kostas Ferles*, Benjamin Mariano*, Shuvendu Lahiri†, Isil Dillig*
*The University of Texas at Austin. {*jon, kferles, bmariano, isil*}*@cs.utexas.edu*
†Microsoft Research. *shuvendu.lahiri@microsoft.com*

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00085

***Abstract*—Smart contracts are programs that run on the blockchain and digitally enforce the execution of contracts between parties. Because bugs in smart contracts can have serious monetary consequences, ensuring the correctness of such software is of utmost importance. In this paper, we present a novel technique, and its implementation in a tool called SMARTPULSE, for automatically verifying temporal properties in smart contracts. SMARTPULSE is the first smart contract verification tool that is capable of checking *liveness properties*, which ensure that "something good" will eventually happen (e.g., "I will eventually receive my refund"). We experimentally evaluate SMARTPULSE on a broad class of smart contracts and properties and show that (a) SMARTPULSE allows automatically verifying important liveness properties, (b) it is competitive with or better than state-of-the-art tools for safety verification, and (c) it can automatically generate attacks for vulnerable contracts.**

I. INTRODUCTION

Smart contracts are programs that run on the blockchain and facilitate multi-party transactions involving monetary exchange. Because bugs in smart contracts can allow attackers to steal money from other users, programming errors in this context have dire security implications [18], [32]. Furthermore, because smart contracts are immutable once deployed on the blockchain, bugs cannot be fixed after deployment. Therefore, it is critical to ensure the correctness of smart contracts before they are deployed on the Blockchain.

Due to the severe consequences of programming errors in this context, recent years have seen significant interest in developing program analysis tools to improve reliability of smart contracts. Generally speaking, efforts in this space fall under two categories: bug finding and verification. Most bug finding techniques look for certain patterns like *reentrancy* that are often highly correlated with security vulnerabilities. On the other hand, verification techniques aim to construct a proof that the contract satisfies a given formal specification.

In this paper, we describe the design and implementation of a new automated verification framework called SMARTPULSE for checking the correctness of smart contracts. In contrast to prior efforts, our approach is not limited to safety and can also check for liveness properties, which require that something good eventually happens. In fact, liveness properties are particularly important in this context because smart contract properties often have the flavor "If a condition is met, then I will eventually get my money". For example, for a smart contract implementing an on-line auction, an important correctness property is that everyone except the highest bidder should get their money back. Intuitively, this is a liveness property because it stipulates that a desirable event, namely the transfer of funds, will eventually happen.

SMARTPULSE is based on three key design principles:

1) **User-friendly and expressive specification language:** SMARTPULSE allows users to specify their properties in a language called SMARTLTL. At its core, SMARTLTL is based on linear temporal logic (LTL), an intuitive and well-established formalism for expressing properties of traces over time. However, SMARTLTL extends standard LTL with additional constructs that make it easier to express correctness properties of smart contracts.
2) **Ability to customize attack models:** SMARTPULSE is parametrized by an environment model that makes it possible to experiment with different attack models. For example, our approach allows users to customize assumptions about how the attacker can interact with a contract through external calls.
3) **Automation and precision:** SMARTPULSE is a fully automated tool based on the counterexample-guided abstraction refinement (CEGAR) paradigm and provides a unified approach for simultaneously searching for proofs and violations. Furthermore, in cases where SMARTPULSE reports an error, it can generate a concrete attack under which the property will be violated.

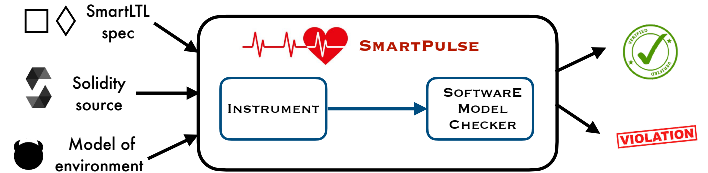

Fig. 1: Schematic workflow of our approach

As shown schematically in Figure 1, SMARTPULSE takes as input a contract P, a SMARTLTL specification φ, and an environment model M (which includes the attack model), and it checks whether P satisfies φ under M. Internally, SMARTPULSE consists of two conceptual phases, namely *program instrumentation* and *software model checking*.

In the first phase, SMARTPULSE instruments the input Solidity program P and generates a new program P' (along with a new property φ') to be fed to the verifier. Our program instrumentation serves three main purposes. First, because the Blockchain can revert the execution of a transaction under certain conditions (e.g., when the contract runs out of "gas"), our instrumentation allows the verifier to be failure-aware. Second, it incorporates the environment model M into the input program. Finally, it generates a pure LTL formula φ' such that P satisfies SMARTLTL specification φ if and only if P' satisfies LTL formula φ'.

Once SMARTPULSE instruments the original contract, it uses software model checking to verify the instrumented program against the LTL specification. Our method builds on prior techniques for LTL property verification [8] and constructs a *Büchi contract* whose feasible, non-terminating paths correspond to LTL property violations. However, in addition, SMARTPULSE leverages domain knowledge about

© 2021, Jon Stephens. Under license to IEEE.
DOI 10.1109/SP40001.2021.00085

smart contracts to make verification more practical. In particular, SMARTPULSE exploits the semantics of SMARTLTL to reduce the number of program paths that it needs to consider, while also utilizing Solidity semantics to simplify its feasibility checking procedure.

We evaluated SMARTPULSE on 191 smart contracts by checking a total of 1947 temporal properties, including both liveness and safety. Our evaluation shows that SMARTPULSE can successfully verify important liveness properties of smart contracts deployed on the Blockchain. We also evaluate SMARTPULSE on safety properties and show that it compares favorably against VERX and KEVM-VER, two state-of-the-art tools for smart contract verification. Finally, we demonstrate that SMARTPULSE can not only detect known vulnerabilities but that it can also generate attacks to exploit those vulnerabilities.

To summarize, this paper makes the following contributions:

- We propose the first toolchain for proving a general class of temporal properties of smart contracts. Notably, our approach can check liveness in addition to safety.
- We propose a language called SMARTLTL for conveniently specifying temporal properties of smart contracts.
- We present a program instrumentation technique that simplifies the subsequent verification problem.
- We show how to incorporate domain knowledge about smart contracts into a software model checker to make verification of temporal properties more tractable.
- We use SMARTPULSE to check 1947 properties across 191 smart contracts and show that SMARTPULSE is effective at verifying/falsifying these properties, while also having the ability to generate attacks.

II. OVERVIEW

Figure 2 shows a Solidity program implementing an auction. This contract has three methods, `bid`, `close`, and `refund`. Users can participate in the auction by calling the `bid` method, which allows each user to place a bid at most once. The `close` method is called by the contract owner when the auction ends, and the `refund` method is also called by the contract owner to refund all losing bids once the auction has closed.

Here, an important correctness property is *"all non-winning bidders should be eventually refunded their bid amount"*. At first glance, this contract seems to satisfy this property because (a) the `bid` function places the refund amount in a mapping called `refunds` every time a participant is outbid, and (b) the `refund` function iterates over this mapping and calls the `transfer` function to send each non-winning bidder their bid amount. However, in reality, this contract does not satisfy this property for two reasons. First, the call to `transfer` implicitly invokes the receiver's so-called *fallback function* which can throw an exception to abort the `refund` transaction. Thus, a malicious user can prevent bidders from receiving their refund. Second, even if there are no malicious users, this contract is still vulnerable because the `refund` method may run of out of gas before all bidders are refunded.

```
1  contract Auction {
2    ...
3    address payable winner = address(0x0);
4    uint currBid = 0; bool closed = false;
5    address payable [] bidders;
6    mapping (address => uint) refunds;
7
8    function bid() payable public {
9      address payable sender = msg.sender;
10     require(!closed && msg.value > currBid));
11     require(refunds[sender] == 0 &&
12             sender != winner);
13     // Store refund of previous winner.
14     refunds[winner] = currBid;
15     // Update winner and currBid.
16     bidders.push(sender);
17     winner = sender;currBid = msg.value;
18   }
19   function close() public onlyOwner {
20     closed = true;
21   }
22   function refund() public onlyOwner {
23     require(closed);
24     for(uint i = 0; i < bidders.length; i++) {
25       uint refAmt = refunds[bidders[i]];
26       refunds[bidders[i]] = 0;
27       bidders[i].transfer(refAmt);
28     }
29   }
30 }
```

Fig. 2: Auction contract. The onlyOwner modifier indicates that `refund/close` can only be called by the auction owner.

A. Usage scenario

The techniques proposed in this paper can uncover such vulnerabilities. To use SMARTPULSE, the user first needs to specify the correctness property as a SMARTLTL formula. For our example, the correctness specification is a liveness property and can be expressed as follows in SMARTLTL:

$$\square\,((\mathsf{finish}(\texttt{bid}, msg.value = X \land msg.sender = L) \land \Diamond\mathsf{finish}(\texttt{close}, L \neq winner)) \rightarrow \Diamond\mathsf{send}(to = L \land amt = X))$$

This property states that if (1) a user L places a bid of amount X (i.e., the first conjunct on the left-hand side of the implication) and (2) L eventually is outbid (the second conjunct), then (3) L will eventually be transferred X amount of ether (the right-hand side of implication). In addition to the property, the user also needs to specify a *fairness constraint* that constrains valid execution traces of the contract. In this case, our fairness constraint states that the auction owner will eventually call the `refund` method:

$$\Diamond\mathsf{start}(\texttt{refund}, closed \land sender = owner)$$

If we invoke SMARTPULSE with this specification, it fails to verify the contract and returns a counterexample trace with contract owner A and three participants B, C, and D. In this trace, user B bids 15 Ether, followed by a bid of 16 Ether from C and 17 Ether from D. Then, A calls the `close` and `refund` methods, where `refund` attempts to transfer 15 Ether back to B, but B's fallback method throws an exception. As a result, A never ends up transferring C her refund.

One way to fix this vulnerability is to change the contract's interface. Specifically, rather than implementing a `refund` method called by the contract owner, the new interface now has a `withdraw` method (shown in Figure 3) to be called by each participant. Since the interface of the contract has changed, we also need to change the fairness constraint to:

$$\Diamond\mathsf{start}(\texttt{withdraw}, closed \land L \neq winner \land L = msg.sender)$$

This says that a losing bidder L will eventually call the `withdraw` function after the auction has closed. With this revised implementation and fairness constraint, SMARTPULSE is now able to verify the correctness of the auction.

```
1  function withdraw() public {
2    require(closed);
3    refund = refunds[msg.sender];
4    refunds[msg.sender] = 0;
5    msg.sender.transfer(refund);
6  }
```

Fig. 3: Replacement of method refund in Figure 2

B. Design Choices behind SMARTPULSE

With this motivating example in mind, we now highlight important design choices behind our approach.

Spec language. Our specification language, SMARTLTL, provides useful predicates like finish, send, start, and revert that allow users to constrain important events occurring during the life of a smart contract. Without such an interface, a user would have to manually instrument the program with auxiliary variables and express the temporal property using pure LTL. By providing a higher-level specification language, SMARTPULSE raises the level of abstraction and automatically performs any required program instrumentation.

Attacks. As our example demonstrates, smart contracts can be vulnerable in subtle ways that only arise when they interact with a malicious contract through its falback methods. To this end, SMARTPULSE can be parameterized with an adversary model that allows users to customize the capabilities of an attacker. Crucially, for vulnerable contracts, SMARTPULSE can also generate concrete attacks, including scenarios that require synthesizing a malicious fallback method.

Reverts. As illustrated in our example, liveness properties can be violated due to exceptions (i.e., "reverts"). Thus, it is crucial to develop a technique that can reason about exceptions.

Gas usage. As we also saw in this example, subtle vulnerabilities may arise to due to not having enough gas to perform some computation. Thus, our technique must also reason about the contract's gas usage.

III. BACKGROUND

In this section we present some background material required for understanding the rest of this paper.

A. Ethereum Virtual Machine

The EVM is an environment in which *Ethereum accounts* interact with each other. Ethereum offers two types of accounts: *external accounts* that are owned by people and *contract accounts* that store and execute code. Every account has a *balance* indicating the amount of Ether (Ethereum's crypto-currency) owned by that account. Accounts interact with each other by issuing *transactions*, which execute code or transfer Ether between accounts.

In EVM, computations are performed by *miners*, who get a fee (measured in *gas*) for performing that computation. Thus, whenever an account issues a transaction, it must provide a certain amount of gas that can be used to pay these miners. If the issuing account does not provide sufficient gas, the transaction is *reverted*, and no modifications to the blockchain are made. Otherwise, the transaction is be *successful*, and all modifications are committed to the blockchain.

B. Solidity Programming Language

Solidity (the most popular programming language for smart contract development) is a statically-typed, object-oriented

Prog $\mathcal{P}$	$\rightarrow$	$\mathcal{C}+$
Con $\mathcal{C}$	$\rightarrow$	$\text{contract CName } \{\ (\mathsf{f} : \tau)^*\ m^+\}$
Func m	$\rightarrow$	$\text{def m}(\overline{v} : \overline{\tau}) = s$
Stmt s	$\rightarrow$	$s_{\mathsf{A}} \mid s; s \mid \text{if } v \text{ then } s \text{ else } s \mid \text{while } v \text{ do } s$
Atom s_{A}	$\rightarrow$	$l \leftarrow e \mid \text{return } v \mid \text{revert} \mid \text{assume } v$
LHS l	$\rightarrow$	$v \mid v.f \mid v[v]$
Expr e	$\rightarrow$	$l \mid c \mid \star \mid v.m(\overline{v}) \mid v.\text{transfer}(v) \mid v \oplus v \mid \oslash v$
Type τ	$\rightarrow$	$\tau_{\mathsf{B}} \mid \tau_{\mathsf{B}} \mapsto \tau \mid \mathcal{C}$
Base τ_{B}	$\rightarrow$	$\text{int} \mid \text{bool} \mid \text{address}$

Fig. 4: Core fragment of Solidity, where $\oplus$ and $\oslash$ indicate binary and unary operators respectively.

language with features targeted to the EVM. In this paper, we model Solidity's key features using the language presented in Figure 4. In particular, a Solidity program consists of a set of contracts, where each contract has a set of fields (called *state variables*) and methods, one of which is a designated *fallback method*. Accounts can invoke the contract's public methods to initiate a transaction that executes the body of that method.[1]

In what follows, we briefly explain Solidity features that are relevant to the rest of this paper.

Types. In addition to int and bool, Solidity provides the address primitive type for uniquely identifying an Ethereum account. Every contract name can be used as a type, and mappings $\tau_{\mathsf{B}} \mapsto \tau$ are first-class citizens.

Revert. The revert construct in Solidity initiates the rollback of a transaction and undoes all side effects of the current transaction on the blockchain.[2] In the remainder of this paper, we use the terms "revert" and "exception" interchangeably.

Ether transfer. Solidity provides multiple constructs for transferring Ether; we model all of these using x.transfer(y), which sends y amount of Ether from this contract to address x. If x is the address of a contract $\mathcal{C}$, then $\mathcal{C}$'s fallback method is executed immediately following the successful transfer of Ether. Note that fallback methods can execute arbitrary code, including calling methods of the contract that initiated the transfer — this is known as *reentrancy*.

Verification constructs. Since we use the same language presented in Figure 4 for program instrumentation, our core language contains two constructs that facilitate verification. First, we use the symbol $\star$ to denote a non-deterministic value. Second, assume v tells the verifier to assume that boolean variable v is true. Note that our assume statement is only for verification purposes and is not the same as the require statement in Solidity, which can be modeled using revert.

External calls. We say that a call is an *external call* if a method from a different program is called, or a call is made to built-in functions like transfer.

EVM-specific features. Solidity passes an implicit EVM-specific parameter called msg that contains information pertinent to the call. For example, msg.sender stores the address of the account that initiated the call.

IV. SPECIFYING PROPERTIES IN SMARTLTL

In this section, we describe the syntax and semantics of our SMARTLTL specification language.

[1] In this paper, we assume all methods are public.

[2] Depending on the Solidity version, reverts can be expressed in multiple ways (e.g., `throw`, `require`, etc.). We model all of these using revert.

$$
\begin{array}{lll}
\text{Spec } \varsigma & \rightarrow & (\varphi, \varphi) \\
\text{LTL } \varphi & \rightarrow & \phi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \\
& & \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi\, \mathcal{U}\, \varphi \\
\text{Atom } \phi & \rightarrow & \mathsf{start}(f, \psi) \mid \mathsf{finish}(f, \psi) \mid \mathsf{revert}(f, \psi) \mid \mathsf{send}(\psi) \\
\text{Pred } \psi & \rightarrow & t \;\mathsf{comp}\; t \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \\
\text{Term } t & \rightarrow & X \mid e \mid t \;\mathsf{op}\; t \\
\text{Exp } e & \rightarrow & v \mid e.f \mid f(\bar{e}) \mid \mathsf{old}(v) \mid \mathsf{csum}(v) \mid \mathsf{fsum}(f, v, \psi)
\end{array}
$$

Fig. 5: Syntax of SMARTLTL.

A. Syntax and Informal Semantics

As shown in Figure 5, a SMARTLTL specification consists of a pair of formulas $(\varphi_\mathsf{F}, \varphi_\mathsf{P})$ where φ_F is a *fairness assumption* and φ_P is the actual temporal property we want to verify. As illustrated in Section II, fairness assumptions are useful for expressing how a rational agent would use the contract (e.g., a user who wants to withdraw their money would invoke the `withdraw` transaction until they receive the fund). Thus, when checking for the property of interest, we can disregard executions where the fairness assumptions are violated.

Both fairness assumptions and correctness properties in SMARTLTL are expressed as LTL-like formulas that contain boolean connectives and temporal operators such as always ($\Box$), eventually ($\Diamond$), next ($\bigcirc$), and until ($\mathcal{U}$). However, unlike standard LTL where the basic building blocks are propositional variables, the building blocks of SMARTLTL are Solidity-specific predicates, which we explain next.

Expressions. SMARTLTL expressions include variables v, field accesses $e.f$, and function calls $f(\bar{e})$ where f is a *pure* function. In addition, SMARTLTL provides a construct of the form $\mathsf{old}(v)$, which refers to the value of a program variable at the beginning of the current transaction. speclang also provides two aggregation constructs, csum and fsum, that we found to be very useful for writing specifications. Given a mapping v, $\mathsf{csum}(v)$ yields the sum of all values stored in v. Similarly, $\mathsf{fsum}(f, v, \psi)$ allows aggregation of function argument values throughout the execution of the contract. That is, $\mathsf{fsum}(f, v, \psi)$ yields the sum of all values provided as argument v of f over all successful invocations satisfying constraint ψ. For instance, $\mathsf{fsum}(f, x_0, x_0 > 0)$ yields the sum of all positive-valued first arguments of `f` throughout the contract's execution.

Terms. SMARTLTL terms include expressions e as well as free (but implicitly-universally quantified) variables X. Terms can be composed using standard arithmetic operators op.

Atomic predicates. The basic building blocks of a SMARTLTL specification are *atomic predicates* ϕ that refer to Solidity events. SMARTLTL provides four key predicates:

1) Predicate $\mathsf{start}(f, \psi)$ is true if a transaction f is started in a context that satisfies predicate ψ.
2) Predicate $\mathsf{finish}(f, \psi)$ evaluates to true if f has successfully finished executing in a state satisfying ψ. We do not consider a transaction to be finished if f reverts.
3) Predicate $\mathsf{revert}(f, \psi)$ is true if transaction f is reverted when started in a context satisfying ψ.
4) Predicate $\mathsf{send}(\psi)$ is true if $\mathsf{transfer}$ is called in a state satisfying ψ.

The last predicate $\mathsf{send}(\psi)$ is actually just syntactic sugar for $\mathsf{start}(\mathsf{transfer}, \psi)$; however, we include it as a separate predicate for convenience.[3] Also, for a predicate $\mathsf{Pred}(f, \psi)$, we allow users to write $\star$ instead of a specific function f. Here, the wildcard symbol $\star$ denotes *any* function, so $\mathsf{Pred}(\star, \psi)$ is equivalent to writing $\bigwedge_{f \in \mathsf{CMethods}} \mathsf{Pred}(f, \psi)$.

Informal semantics. To understand SMARTLTL semantics, we first need a notion of "time step". Since SMARTLTL predicates talk about starting and finishing transactions, we consider the clock to "tick" every time an external function is called or that function returns/reverts. Thus, the next time step indicates the next call/return/revert event from an external function, which includes built-in functions as well as methods defined by a different contract. Under this notion of "time step", SMARTLTL operators have the following semantics:

- The next operator $\bigcirc$ expresses that a predicate is true in the next time step.
- The always operator $\Box$ captures that a property holds globally (i.e., every time an external function is called).
- The eventually operator $\Diamond$ expresses that the property will hold at some point in the future.
- The until operator $\mathcal{U}$ expresses that its first argument continues to be true until the second argument becomes true.

Formal semantics. We formalize SMARTLTL semantics in terms of *execution traces*. For the purposes of this paper, an execution trace τ is a sequence of triples of the form (f, κ, σ) where f is the name of an external function, $\kappa \in \{\mathsf{call}(\overline{x}), \mathsf{return}, \mathsf{revert}\}$, and σ is a *valuation* mapping SMARTLTL terms to their values. Given a SMARTLTL formula φ and a trace τ, we write $\tau \models \varphi$ (resp. $\tau \not\models \varphi$) to denote that φ evaluates to true (resp. false) under trace τ. A complete formalization of this relation is in Appendix A.

Given a SMARTLTL specification $\varsigma \equiv (\varphi_\mathsf{F}, \varphi_\mathsf{P})$ consisting of fairness assumptions φ_F and property φ_P, we say that an execution trace τ satisfies ς, denoted $\tau \models \varsigma$, if and only if (a) either $\tau \not\models \varphi_\mathsf{F}$, or (b) $\tau \models \varphi_\mathsf{P}$. Then, given a Solidity contract $\mathcal{P}$ and SMARTLTL specification ς, we say that $\mathcal{P}$ satisfies the specification, written $\mathcal{P} \models \varsigma$, if, for all feasible execution traces τ of $\mathcal{P}$, we have $\tau \models \varsigma$.

Example IV.1. Consider a crowdsale that allows users to invest in a beneficiary by buying tokens. An important correctness property for such a crowdsale is that the beneficiary will eventually receive all funds that have been invested. We express this property as $(\varphi_\mathsf{F}, \varphi_\mathsf{P})$ shown in Figure 6. The fairness property (φ_F) specifies that if the crowdsale closes, the beneficiary, b, will eventually try to `withdraw` the invested funds. The temporal property (φ_P) specifies that if the crowdsale eventually closes successfully (i.e., the sum of the $investments$ is greater than the fundraising $goal$ at closing), then eventually the beneficiary will be sent the total sum of tokens bought using the `buy` function, after which no more ether is sent to the beneficiary.

V. SPECIFYING ENVIRONMENT MODELS

In addition to the contract's source code and a SMARTLTL specification, SMARTPULSE also takes as input an *environment model* $\mathcal{E} = (\mathcal{A}, \mathcal{B})$ where $\mathcal{A}$ models the attacker and $\mathcal{B}$ specifies assumptions about the Blockchain execution environment. By default, SMARTPULSE provides a standard Blockchain

[3]The semantics of $\mathsf{send}(\psi)$ is *on purpose* not $\mathsf{finish}(\mathsf{transfer}, \psi)$ because we can never guarantee that send will successfully finish, as the receiver's fallback method is free to call revert.

$$\varphi_{\mathsf{F}} : \Box\,(\mathsf{finish}(\texttt{close}) \rightarrow \Diamond\mathsf{start}(\texttt{withdraw}, msg.sender = b))$$
$$\varphi_{\mathsf{P}} : \Diamond(\mathsf{finish}(\texttt{close}, \mathsf{csum}(investments) \geq goal)) \;\rightarrow$$
$$\Diamond\big(\mathsf{send}(amount = \mathsf{fsum}(\texttt{buy}, msg.value, true) \wedge to = b) \;\wedge\; \circ\Box\,(\neg\mathsf{send}(to = b))\big)$$

Fig. 6: Specification for crowdsale contract from Example IV.1

model that need not be modified by users, as this model only changes due to modifications to the EVM or Solidity compiler. SMARTPULSE also provides three default attacker models, which can be further customized by users.

Specifying attacker model. Formally, an attacker model is a triple $\mathcal{A} = (\sigma_0, \mathcal{F}, \eta)$ where:

- σ_0 specifies the initial Blockchain state as a mapping from contract instances $\mathcal{C}$ to symbolic values describing their initial state. We assume that the attacker can only invoke transactions defined by those contract instances in $\mathcal{C}$.
- $\mathcal{F} \subseteq \mathsf{CMethods}(\mathcal{C})$ is the set of functions that may be invoked by the attacker (e.g., through fallback methods).
- $\eta \in \mathbb{N} \cup \{\infty\}$ is the maximum number of contract methods that the attacker can invoke.

While the user is free to explicitly define her own attacker model, SMARTPULSE provides three default models that share the same initial state σ_0 but differ in $\mathcal{F}$ and η. For all three default models, the initial state consists of a single instance of the contract to be analyzed, and the initial state of the contract is determined by symbolically executing the constructor on symbolic inputs.[4] We now explain how the three default attacker models differ from each other.

1) ***No reentrancy.*** For our most restrictive attacker model, we have $\mathcal{F} = \emptyset$ and $\eta = 0$. This model corresponds to the assumption that there is no re-entrancy and can be useful for analyzing contracts that satisfy the *effective external callback freedom (EECF)* assumption [27].
2) ***Single callback.*** For this model, we have $\eta = 1$ and $\mathcal{F} = \mathsf{CMethods}(\mathcal{C})$ where $\mathcal{C}$ is the set of contract instances in σ_0. In other words, this model allows an attacker to call any contract method a single time, which is equivalent to Zeus' modeling of fallback functions [17].
3) ***Powerful adversary.*** For our least restrictive model, we have $\eta = \infty$ and $\mathcal{F} = \mathsf{CMethods}(\mathcal{C})$. This allows an attacker to invoke any of the contract's public functions arbitrarily many times.

Blockchain model. In addition to the attacker model, SMARTPULSE also utilizes a so-called Blockchain model that specifies assumptions about the contract's execution environment. Formally, a Blockchain model is a quadruple $\mathcal{B} = (\mathcal{G}, \mathsf{g_{min}}, \mathsf{g_{max}}, \mathsf{g_{transfer}})$ where:

- $\mathcal{G} : \mathsf{Stmt} \rightarrow \mathbb{N}$ provides a cost model for the gas usage of each statement.
- $\mathsf{g_{min}}$ (resp. $\mathsf{g_{max}}$) $\in \mathbb{N}$ is the minimum (resp. maximum) gas allowance a transaction can be provided.
- $\mathsf{g_{transfer}} \in \mathbb{N}$ is the amount of gas provided to a transfer.

Because the Blockchain model is not contract-specific, a single model is sufficient for analyzing any contract using a specific Solidity compiler and a specific EVM version. Thus, in practice, users do not need to worry about configuring SMARTPULSE's Blockchain model.

[4]If the contract to be analyzed involves auxiliary contracts, then the initial state includes an instance of the auxiliary contracts as well.

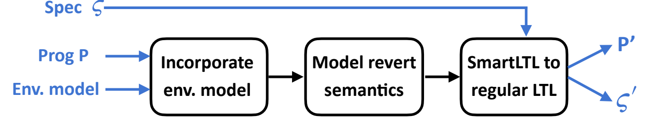

Fig. 7: Program instrumentation pipeline

VI. PROGRAM INSTRUMENTATION

To verify a Solidity contract against a SMARTLTL specification, our method performs a sequence of three program transformations, as shown in Figure 7. First, we incorporate the environment model directly into the program. Next, we perform an instrumentation that accurately models the semantics of revert by introducing so-called *history variables*. Finally, we further instrument the program with boolean variables and convert the original SMARTLTL specification to standard LTL. Next, we explain these transformations in more detail.

A. Modeling Environment

To incorporate the environment model $\mathcal{E}$ into a Solidity program, we need to (1) model the attacker by introducing non-deterministic fallback methods and explicitly invoking them; (2) track the contract's gas usage by introducing a new variable; (3) generate a *harness contract* that simulates the usage of the target contract.

Modeling the attacker. We model the attacker by creating the transfer stub shown in Figure 8. Here, our stub first checks whether the sender's balance is sufficient to perform the transfer (lines 2-3) and reverts if it is not. Otherwise, it adjusts the sender's and receiver's balances accordingly (lines 4-5) and finally calls the receiver account's fallback method (line 6).

The attacker model $\mathcal{A} = (\sigma_0, \mathcal{F}, \eta)$ comes into play when generating an implementation of the receiver's fallback method. As shown in Figure 8, our model performs up to η iterations (line 10), and, in each iteration, it picks a contract c under analysis and one of the methods $m \in \mathcal{F}$ that the attacker is allowed to call under $\mathcal{A}$. Then, it invokes $c.m$ with arbitrary arguments (line 13) and reverts if any of the method calls within the fallback are also reverted.[5]

Modeling gas usage. In practice, important properties (esp. those involving liveness) may be violated when a contract runs out of gas. Thus, our instrumentation introduces an auxiliary variable gas and explicitly tracks gas usage.

Figure 9 presents our instrumentation (for relevant Solidity statements) using the notation $s \rightsquigarrow s'$, which indicates that s' is the instrumented version of s. For an atomic statement s_{A}, we decrement the gas variable by $\mathcal{G}(s_{\mathsf{A}})$, and, if the value of gas becomes negative, we revert the current transaction.

The next rule in Figure 9 shows how to perform gas instrumentation for method calls. Here, we generate two statements, namely ι_{pre} and ι_{post}, that precede and succeed the original

[5]Here, for simplicity, we assume that the return value of a method indicates whether it was reverted. In reality, our instrumentation uses the history variable described in Section VI-B to determine whether a method call was reverted.

```
1  def transfer(amt : int) =
2     senderBal ← msg.sender.balance;
3     if senderBal < amt then revert;
4     this.balance ← this.balance + amt;
5     msg.sender.balance ← senderBal - amt;
6     fallback();
7
8  def fallback() =
9     i ← 0;
10    while i < η ∧ ⋆ do
11       c ← choose(𝒞);
12       m ← choose(ℱ);
13       if c.m(⋆̄) then revert;
14       i ← i + 1;
```

Fig. 8: Modeling semantics of transfer and fallback behavior

$$\text{(Stmt.)}\quad \frac{\iota \equiv (\mathsf{gas} \leftarrow \mathsf{gas} - \mathcal{G}(s_{\mathsf{A}}); \text{if gas } < 0 \text{ then revert})}{s_{\mathsf{A}} \rightsquigarrow \iota; s_{\mathsf{A}}}$$

$$\text{(Call)}\quad \frac{\iota_{pre} \equiv (\mathsf{v.gas} \leftarrow \mathsf{gas} - \mathcal{G}(\mathsf{s})) \qquad \iota_{post} \equiv (\mathsf{gas} \leftarrow \mathsf{v.gas})}{s \equiv l \leftarrow \mathsf{v.m}(\overline{\mathsf{v}}) \rightsquigarrow \iota_{pre}; s; \iota_{post}}$$

$$\text{(Transfer)}\quad \frac{\begin{array}{c}\iota_{pre} \equiv (\mathsf{to.gas} \leftarrow \min(\mathsf{gas}, \mathsf{g}_{\mathsf{transfer}}); \mathsf{t} \leftarrow \mathsf{to.gas}) \\ \iota_{post} \equiv (\mathsf{gas} \leftarrow \mathsf{gas} - (\mathsf{t} - \mathsf{to.gas}))\end{array}}{s \equiv l \leftarrow \mathsf{to.transfer}(\mathsf{v}) \rightsquigarrow \iota_{pre}; s; \iota_{post}}$$

Fig. 9: Gas usage instrumentation.

statement respectively. Statement ι_{pre} forwards the remaining gas (minus the cost of performing the call) from the current contract to the receiver contract. Conversely, ι_{post} retrieves the remaining gas after the call returns. The last rule for transfer is similar to the one for method calls, but, instead of forwarding all the remaining gas to transfer, it only forwards up to $\mathsf{g}_{\mathsf{transfer}}$ of gas (statement ι_{pre}). Any unused gas after the call to transfer is claimed by the current contract.

Harness. Smart contracts are open programs, so we need to generate a harness that models all possible ways that accounts may interact with this contract. As shown in Figure 10, the harness created by SMARTPULSE first initializes the contract state according to the environment model (line 3). It then performs an arbitrary number of iterations where it (1) picks a random contract c and one of its methods m (lines 6-7), (2) updates auxiliary variables (e.g, `msg`, `block`,) (lines 8 and 9) and, (3) begins a transaction by providing arbitrary parameters to the selected method (line 10).

B. Modeling Semantics of Revert Statements

Since our specification language requires differentiating between successful and failed transactions, we need to precisely model the semantics of revert statements. We do this by instrumenting the program with so-called *history variables* [1].

At a high level, the idea is as follows: First, we introduce a history variable called fail that tracks whether the current transaction is reverted. Then, for each function f, we create

```
1  def harness() =
2    aux ← initBlockchainVars();
3    assume    ⋀          ψ_i
          (c_i,ψ_i)∈σ_0
4
5    while(true) do
6      c ← choose(Contracts(σ_0));
7      m ← choose(CMethods(c));
8      aux ← updateBlockchainVars();
9      c.gas ← ⋆;assume g_min ≤ c.gas ∧ c.gas ≤ g_max;
10     BeginTX(aux, c.m(⋆̄));
```

Fig. 10: Harness generated by SMARTPULSE.

```
1  contract C {
2    i : int;
3    def foo(j : int) = i ← i + j;
4                       if (i > 100) then revert
5                       else return i
6  }
```

(a) Original Contract.

```
1  contract C {
2    fail : bool;
3    i : int;i_✗ : int;
4
5    def foo_✓(j : int) = i ← i + j;
6                          if (i > 100) then
7                            fail ← true;return ⋆
8                          else return i
9    def foo_✗(j : int) = i_✗ ← i_✗ + j;
10                         if (i_✗ > 100) then
11                           fail ← true;return ⋆
12                         else return i_✗
13   def foo(j : int) = fail ← false;
14                      if (⋆) then
15                        r ← foo_✓(j);
16                        assume  ¬ fail
17                      else
18                        i_✗ ← i;r ← foo_✗(j);
19                        assume fail
20                      return r
21 }
```

(b) Instrumented Contract.

Fig. 11: Toy Contract and its instrumented version.

two copies $f_{✓}$ and $f_{✗}$, where $f_{✓}$ (resp. $f_{✗}$) represents the version of f that is called in a successful (resp. failing) execution. The bodies of $f_{✓}$ and $f_{✗}$ are identical except that (a) $f_{✗}$ uses a shadow variable $v_{✗}$ instead of state variable v, and (b) a call to function g in the original function is replaced by a call to $g_{✓}$ in $f_{✓}$ and by $g_{✗}$ in $f_{✗}$. Finally, the original function f is replaced by a wrapper that "guesses" whether the transaction is going to fail and then constrains the value of the history variable (using assume statements) to ensure that its "prediction" was correct. Intuitively, the reason we need to create two copies of each method is that one copy (i.e., $f_{✓}$) affects blockchain state, while the other one (i.e., $f_{✗}$) does not.

We illustrate this idea using the example in Figure 11, where the original code reverts if `i` exceeds 100. We highlight the following salient features of our instrumentation:

- The instrumented contract contains the history variable fail, which is initialized to false (line 13 in Figure 11b), and the call to revert in the original program (line 4 in 11a) is modeled as setting fail to true and then returning immediately (lines 7 and 11 in Figure 11b).
- The instrumented contract contains two copies of foo that are identical except that $\mathsf{foo}_{✗}$ uses shadow variable $i_{✗}$.
- The wrapper function foo initializes fail to false and then non-deterministically calls either $\mathsf{foo}_{✗}$ or $\mathsf{foo}_{✓}$. Before the call to $\mathsf{foo}_{✗}$, we initialize the shadow state variable $i_{✗}$ to i.
- After the call to $\mathsf{foo}_{✓}$, we add a statement assume(¬fail) to ensure that $\mathsf{foo}_{✓}$ is only called during non-failing transactions. For the same reason, we also add an assumption that fail is true after the call to $\mathsf{foo}_{✗}$.

Here, note that the value of state variable i is unchanged in failing executions since $\mathsf{foo}_{✗}$ operates on shadow variable $i_{✗}$. Furthermore, the instrumented assume statements enforce that failing executions only call $\mathsf{foo}_{✗}$ whereas successful executions only call $\mathsf{foo}_{✓}$. Since any program path where an assumption is violated is considered infeasible by the verifier, this instrumentation allows us to faithfully and precisely capture the semantics of revert.

Figure 12 presents this instrumentation more formally.

$$\text{(Revert)}\quad \frac{\iota \equiv (\mathsf{fail} \leftarrow \mathsf{true};\ \mathsf{return}\ \star)}{\mathcal{C}, \varepsilon \vdash \mathsf{revert} \hookrightarrow \iota}$$

$$\text{(Call)}\quad \frac{\iota\ \equiv\ (l \leftarrow \mathsf{v}.m_\varepsilon(\overline{\mathsf{v}}); \mathsf{if\ fail\ then\ return}\ \star)}{\mathcal{C}, \varepsilon \vdash l \leftarrow \mathsf{v.m}(\overline{\mathsf{v}}) \hookrightarrow \iota} \qquad \text{(Stmt ✗)}\quad \frac{\varepsilon = ✗ \qquad \overline{f} = \text{Fields}(\mathcal{C})}{\mathcal{C}, \varepsilon \vdash s_{\mathsf{A}} \hookrightarrow s_{\mathsf{A}}[\overline{f_{✗}}/\overline{f}]}$$

$$\text{(If)}\quad \frac{\mathcal{C}, \varepsilon \vdash s_1 \hookrightarrow s_1' \qquad \mathcal{C}, \varepsilon \vdash s_2 \hookrightarrow s_2'}{\mathcal{C}, \varepsilon \vdash \mathsf{if}\ v\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \hookrightarrow \mathsf{if}\ v\ \mathsf{then}\ s_1'\ \mathsf{else}\ s_2'} \qquad \text{(While)}\quad \frac{\mathcal{C}, \varepsilon \vdash s \hookrightarrow s'}{\mathcal{C}, \varepsilon \vdash \mathsf{while}\ v\ \mathsf{do}\ s \hookrightarrow \mathsf{while}\ v\ \mathsf{do}\ s'}$$

$$\text{(Method)}\quad \frac{\mathcal{C}, \varepsilon \vdash s \hookrightarrow s'}{\mathcal{C}, \varepsilon \vdash\ \mathsf{def\ m}(\overline{\mathsf{v}} : \overline{\tau}) = s \hookrightarrow \mathsf{def\ m}_\varepsilon(\overline{\mathsf{v}} : \overline{\tau}) = s'} \qquad \text{(Seq)}\quad \frac{\mathcal{C}, \varepsilon \vdash s_1 \hookrightarrow s_1' \qquad \mathcal{C}, \varepsilon \vdash s_2 \hookrightarrow s_2'}{\mathcal{C}, \varepsilon \vdash s_1; s_2 \hookrightarrow s_1'; s_2'}$$

$$\text{(Contract)}\quad \frac{\mathcal{C} \vdash m_i \mapsto m_i', m_{✓}, m_{✗} \qquad \iota \equiv\ \mathsf{fail} : \mathsf{bool}}{\mathsf{contract\ C}\ \{\overline{f}\ \overline{m}\} \hookrightarrow \mathsf{contract\ C}\ \{\iota\ \overline{f}\ \overline{f_{✗}}\ \overline{m_{✓}}\ \overline{m_{✗}}\ \overline{m'}\}}$$

$$\text{(Wrapper)}\quad \frac{\begin{array}{c}\overline{f} = \text{Fields}(\mathcal{C}), \quad \mathcal{C}, ✓ \vdash m \hookrightarrow m_{✓}, \quad \mathcal{C}, ✗ \vdash m \hookrightarrow m_{✗} \\ \iota_s \equiv (\mathsf{r} := m_{✓}(\overline{\mathsf{v}}); \mathsf{assume}\ \neg\mathsf{fail}; \mathsf{return\ r};) \qquad \iota_{\mathsf{f}} \equiv (\overline{f_{✗}} := \overline{f}; \mathsf{r} := m_{✗}(\overline{\mathsf{v}}); \mathsf{assume\ fail}; \mathsf{return\ r};)\end{array}}{\mathcal{C} \vdash m \mapsto \mathsf{def}\ m(\overline{\mathsf{v}} : \overline{\tau}) = \mathsf{fail} \leftarrow \mathsf{false}; \mathsf{if}\ \star\ \mathsf{then}\ \iota_{\mathsf{s}}\ \mathsf{else}\ \iota_{\mathsf{f}}, m_{✓}, m_{✗}}$$

Fig. 12: Instrumentation for Solidity's exception semantics. For statements that are not shown, we assume $\mathcal{C}, \varepsilon \vdash s \hookrightarrow s$.

Specifically, we use two types of judgments: (1) The judgment $\mathcal{C}, \varepsilon \vdash s \hookrightarrow s'$ indicates that statement s in $\mathcal{C}$ is re-written into s' in context $\varepsilon \in \{✓, ✗\}$ indicating whether this is a failing or successful execution. (2) $\mathcal{C} \vdash m \mapsto m', m_{✓}, m_{✗}$ indicates that method m in contract $\mathcal{C}$ is re-written into three variants, where $m_{✓}$ (resp. $m_{✗}$) is the variant that is called in successful (resp. failing) executions, and m' is the wrapper method that ensures that $m_{✓}$ (resp. $m_{✗}$) is only called during successful (resp. failing) executions.

We now explain the rules from Figure 12 in more detail.

Revert. Our re-write rules eliminate revert statements by assigning the history variable fail to true and then returning.

Stmt ✗. When rewriting atomic statements in context ✗, we replace any state variable v with its shadow version $v_{✗}$.

Call. We re-write function calls by (a) calling the correct (i.e., successful or failing) version of the function depending on context $\varepsilon \in \{✓, ✗\}$, and (b) propagating any exceptions that occur. For instance, consider the following method:

$$\mathsf{def\ bar}(\mathsf{k} : \mathsf{int}) = \mathsf{j} \leftarrow \mathsf{foo}(\mathsf{k}); \mathsf{return\ j};$$

Then, $\mathsf{bar}_{✓}$ and $\mathsf{bar}_{✗}$ would be as follows:

$$\mathsf{def\ bar}_{✓}(\mathsf{k} : \mathsf{int}) = \mathsf{j} \leftarrow \mathsf{foo}_{✓}(\mathsf{k}); (\mathsf{if\ fail\ then\ return}\ \star); \mathsf{return\ j};$$
$$\mathsf{def\ bar}_{✗}(\mathsf{k} : \mathsf{int}) = \mathsf{j} \leftarrow \mathsf{foo}_{✗}(\mathsf{k}); (\mathsf{if\ fail\ then\ return}\ \star); \mathsf{return\ j};$$

Wrapper. This rule synthesizes a wrapper for method m. Since m serves as the entry point of a transaction, we initialize fail to false and then predict whether the transaction will succeed or fail by non-deterministically callling $m_{✓}$ and $m_{✗}$. The assumptions added after the call ensure that we made the right prediction.

Contract. This rule instruments the whole contract by introducing a history variable fail and shadow fields. It also generates the wrapper, successful, and failed methods for every method of the original contract.

C. Converting SmartLTL Specifications to Regular LTL

To build upon existing verification techniques for checking temporal properties, the specifications must be expressed in standard LTL. Toward this goal, we show how to instrument the program so that the verification problem can be expressed as checking a standard LTL property over the instrumented program. Specifically, our method consists of three steps: First, we introduce variables that store values of SMARTLTL expressions $\mathsf{old}(v), \mathsf{csum}(v), \mathsf{fsum}(f, i, \psi)$. Second, for each atomic SMARTLTL predicate ϕ (e.g., $\mathsf{start}(f, \psi)$, we introduce a fresh boolean variable b and rewrite the SMARTLTL specification into a pure LTL specification by replacing every occurrence of ϕ with b. Finally, we instrument the program to make the correct assignments to these boolean variables.

Since the last step is non-trivial, we describe it in more detail in Figure 13 using judgments $\Gamma \vdash \mathcal{P}[s] \xrightarrow{\phi} \mathcal{P}[s']$. Here, Γ maps SMARTLTL expressions to program variables

START-TRUE

$$\frac{\begin{array}{c} s \equiv (v \leftarrow x.f(\overline{y})) \quad \phi \equiv \mathsf{started}(f, \psi) \\ \mathsf{BoolVar}(\phi, b) \quad \psi' = \mathsf{Sub}(\psi, \Gamma)\end{array}}{\Gamma \vdash \mathcal{P}[s] \xrightarrow{\phi} \mathcal{P}[b \leftarrow \psi'; s]}$$

START-FALSE

$$\frac{s \equiv (\mathsf{def} f = s') \quad \phi \equiv \mathsf{started}(f, \psi) \quad \mathsf{BoolVar}(\phi, b)}{\Gamma \vdash \mathcal{P}[s] \xrightarrow{\phi} \mathcal{P}[\mathsf{def} f = (b \leftarrow \mathsf{false}; s')]}$$

FINISH-TRUE

$$\frac{\begin{array}{c} \mathsf{Return}(s) \quad \mathsf{InWrapper}(s, f) \\ \phi \equiv \mathsf{finish}(f, \psi) \quad b = \mathsf{BoolVar}(\phi) \quad \psi' = \mathsf{Sub}(\psi, \Gamma)\end{array}}{\Gamma \vdash \mathcal{P}[s] \xrightarrow{\phi} \mathcal{P}[b \leftarrow \neg \mathit{fail} \wedge \psi'; s]}$$

REVERT-TRUE

$$\frac{\begin{array}{c} \mathsf{Return}(s) \quad \mathsf{InWrapper}(s, f) \\ \phi \equiv \mathsf{revert}(f, \psi) \quad b = \mathsf{BoolVar}(\phi) \quad \psi' = \mathsf{Sub}(\psi, \Gamma)\end{array}}{\Gamma \vdash \mathcal{P}[s] \xrightarrow{\phi} \mathcal{P}[b \leftarrow \mathit{fail} \wedge \psi'; s]}$$

FINISH/REVERT-FALSE

$$\frac{\begin{array}{c} \phi \equiv R(f, \psi) \quad R \in \{\mathsf{finish}, \mathsf{revert}\} \\ s \equiv (v \leftarrow x.f(\overline{y})) \quad b = \mathsf{BoolVar}(\phi)\end{array}}{\Gamma \vdash \mathcal{P}[s] \xrightarrow{\phi} \mathcal{P}[s; b \leftarrow \mathsf{false}]}$$

Fig. 13: SmartLTL to LTL instrumentation. Γ is a mapping from SMARTLTL expressions to their corresponding program variables. Also, $\ominus \in \{\neg, \bigcirc, \Box, \Diamond\}$, $\circledast \in \{\wedge, \vee, \Rightarrow, \mathcal{U}, \mathcal{R}\}$. The function $\mathsf{Sub}(\psi, \Gamma)$ substitutes each SMARTLTL expression e in ψ with its corresponding variable $\Gamma(e)$

```
1  contract Auction {
2    L : address;
3    finish : bool;
4    fsum : int;
5    old : int;
6    ...
7    def bid() = ...
8                if (*)
9                  if(msg.sender = L)
10                   fsum ← fsum + msg.value;
11                 call bid✓();
12                 assume ¬fail
13               ...
14   def withdraw() = old ← L.bal;
15                       ...
16                       finish ← ¬fail ∧
17                                (L.bal-old)=fsum;
18                       return
19   def harness() = L ←⋆;
20                     finish ← false;
21                     fsum ← 0;
22                     old ←⋆;
23                     ...
24                     while(true) do
25                       ...
26                       if(*)
27                         BeginTX(aux, c.withdraw(⋆̄));
28                         finish ← false;
29                       ...
30 }
```

Fig. 14: Instrumented Auction Example.

introduced in step (1).[6] The meaning of this judgment is that statement s in program $\mathcal{P}$ is rewritten to s' when performing instrumentation for predicate ϕ. There are two rules associated with each predicate (labeled PRED-TRUE and PRED-FALSE) for assigning the corresponding boolean variable to true and false respectively.

Start. A boolean b associated with $\mathsf{start}(f, \psi)$ is set to true at a call site of function f if the formula ψ evaluates to true. However, since ψ may contain SMARTLTL expressions, we first obtain a new predicate ψ' by substituting all SMARTLTL expressions with their corresponding program variable stored in Γ. Thus, our instrumentation assigns b to ψ' immediately before an invocation of function f. Futhermore, since b should only be true at call sites and no where else (recall SMARTLTL semantics from Section A), we immediately set b to false as soon as we start executing function f.

Finish. For predicate $\mathsf{finish}(f, \psi)$, we set its corresponding boolean b if its condition ψ holds and if the transaction has not reverted. Since this predicate should only evaluate to true at return points of f and nowhere else, we again set the corresponding boolean variable to false after f's invocation.

Revert. The instrumentation for revert is very similar to finish except that it is assigned to true if the transaction has reverted and its condition is satisfied.

Example VI.1. Recall the auction from Figure 2 that allows users to place a bid at most once. Suppose we extend the fixed version of this auction to allow users to place multiple bids. We would like to ensure that we still correctly refund the losers. The following SMARTLTL property (along with the fairness property stated in Section II) asserts that a loser is eventually refunded the sum of their bids.

$$\Diamond \mathsf{finish}(\texttt{withdraw}, L.bal - \mathsf{old}(L.bal) = \mathsf{fsum}(\mathsf{bid}, msg.val, msg.sender = L))$$

[6] Γ also maps free variables in ϕ to non-deterministic values.

1: **procedure** VERIFY($\mathcal{P}, \varphi_\mathsf{F}, \varphi_\mathsf{P}$)
2: **input:** Program $\mathcal{P}$
3: **input:** φ_F, the fairness constraint
4: **input:** φ_P the property to verify
5: **Output:** Verified, Counterexample or Unknown
6: $\mathcal{A}_\varphi := \mathsf{ConstructBuchi}(\varphi_F \land \neg\varphi_P)$
7: $\mathcal{B} := \mathcal{P} \times \mathcal{A}_\varphi$
8: **while** $\mathcal{L}(\mathcal{B}) \neq \emptyset$ **do**
9: $\pi \in \mathcal{L}(\mathcal{B})$
10: **if** $\mathsf{feasible}(\pi) = \checkmark$ **then**
11: **return** π
12: **else if** $\mathsf{feasible}(\pi) = ?$ **then**
13: **return** Unknown
14: $\mathcal{B} := \mathcal{B} \setminus \pi$
15: **return** Verified

Algorithm 1: Verification framework adapted from [8]

Figure 14 shows snippets of the instrumented auction. Lines 2–5 introduce the variables L for the unbound variable L, finish for the finish atom, fsum for the fsum expression, and old for the $\mathsf{old}(L.balance)$ expression. Lines 19–22 initialize these variables to a non-deterministic value ($\star$), false, 0, and $\star$ respectively. Lines 9–10 instrument the calculation of the fsum expression, line 14 instruments the old expression, and lines 16, 17, and 28 instrument the finish predicate. With this instrumentation, the SMARTLTL formula is translated to $\Diamond$finish in standard LTL.

VII. VERIFICATION ALGORITHM

In this section, we describe our technique for verifying smart contracts against LTL specifications. Our verification algorithm is an instance of the counterexample-guided abstraction refinement (CEGAR) paradigm commonly used for temporal property checking [4], [5], [8], but it exploits domain-specific knowledge about smart contracts to make verification more efficient. In the remainder of this section, we first give an overview of the basic framework we build upon (Section VII-B) and then discuss our domain-specific adaptations in Sections VII-C and VII-D. [7]

A. Background on Büchi Automata

Like all verification techniques for temporal property checking, our approach requires converting the LTL specification to a *Büchi automaton*:

Definition VII.1. (Büchi automaton.) A Büchi automaton $\mathcal{A} = (\Sigma, \mathcal{Q}, q_0, \rightarrow, \mathcal{F})$ is a finite-state automaton that accepts infinite words. Specifically, Σ denotes a finite alphabet, $\mathcal{Q}$ is a finite set of states with initial state q_0 and final (accepting) states $\mathcal{F} \subseteq \mathcal{Q}$. The transition relation $\rightarrow$ is a function $\mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$. The automaton accepts a word $w \in \Sigma^*$ if a run of w on $\mathcal{A}$ visits a set of final states infinitely many times.

Note that every LTL formula can be converted into an equivalent Büchi automaton using standard techniques [10].

B. Overview

As mentioned earlier, our algorithm is an instance of the CEGAR framework for temporal property checking proposed

[7] As we show in Appendix C, these domain-specific adaptations are quite important for making this approach practical.

in prior work [8]. In this section, we give an overview of this framework and then discuss our domain-specific adaptations in the next two subsections.

The basic CEGAR framework for LTL property checking is presented in Algorithm 1 and works as follows: First, it converts the LTL formula $\varphi_F \wedge \neg\varphi_P$ to a Büchi automaton whose language corresponds to words that satisfy the fairness constraint but violate the correctness property. It then constructs another Büchi automaton $\mathcal{B}$ which represents the program together with the specification (line 7). In particular, this automaton is constructed in such a way that the emptiness of $\mathcal{B}$'s language constitutes a proof that $\mathcal{P}$ satisfies the specification. Then, in each iteration of the loop (lines 8–14), the algorithm finds a word (i.e., infinite path) π that is accepted by $\mathcal{B}$ (line 9) and checks whether it is actually feasible under the program semantics. If so, π is a genuine counterexample (line 11), and, if not, the algorithm removes this spurious word π from $\mathcal{B}$ (line 14) and moves on to the next word accepted by $\mathcal{B}$. On the other hand, if the feasibility of π cannot be determined, the algorithm returns unknown (line 13).

Based on the above discussion, there are two key issues we need to address to adopt this verification approach:

1) How to construct a Büchi automaton $\mathcal{B}$ that represents an embedding of the specification into the smart contract;
2) How to determine feasibility of an infinite path π

In the remainder of this section, we address these questions and show how to leverage domain-specific knowledge to simplify the resulting verification problem.

C. Büchi Contracts

We define a *Büchi contract* to represent a smart contract together with its specification:

Definition VII.2. (Büchi contract) Let $\mathcal{P}$ be a smart contract with statements $\mathcal{S}$, program locations $\mathcal{L}$, entry location ℓ_0 and a transition relation $\mathcal{T} : \mathcal{L} \times \mathcal{S} \rightarrow \mathcal{L}$. Given a specification automaton $\mathcal{A}_\varphi = (\Sigma, \mathcal{Q}, q_0, \rightarrow, \mathcal{F})$, the Büchi contract $\mathcal{P} \times \mathcal{A}_\varphi$ is a Büchi automaton $\mathcal{B} = (\widehat{\Sigma}, \widehat{\mathcal{Q}}, \widehat{q_0}, \twoheadrightarrow, \widehat{\mathcal{F}})$ where:

- $\widehat{\Sigma} = \{s;\ \mathsf{assume}(\phi) \mid s \in \mathcal{S} \wedge \phi \in \Sigma\}$
- $\widehat{\mathcal{Q}} = \mathcal{L} \times \mathcal{Q}$ and $\widehat{q_0} = (\ell_0, q_0)$
- The transition relation $\twoheadrightarrow$ maps a state (ℓ_1, q_1) to another state (ℓ_2, q_2) on label $(s; \mathsf{assume}(\phi))$ iff $(\ell_1, s, \ell_2) \in \mathcal{T}$ and $(q_1, \phi, q_2) \in\rightarrow$.
- $\mathcal{F} = \{(\ell, q) \mid q \in \mathcal{F}\ \wedge\ (\mathsf{ExtCall}(\ell) \vee \mathsf{ExtReturn}(\ell))\}$

In this definition, $\mathsf{ExtCall}(\ell)$ (resp. $\mathsf{ExtReturn}(\ell)$) is true if ℓ corresponds to a program location immediately preceding a call to (resp. return from) an external function.[8] Thus, our Büchi contracts are very similar to the notion of *Büchi product programs* from prior work [8] but differ in the construction of the final states. In particular, since the SMARTLTL semantics are defined over calls to and returns from external functions, only those locations that correspond to external call/return points are marked as final states. This difference is quite important for making verification practical since it significantly reduces the number of words (i.e., program paths) accepted by the Büchi contract and allows disregarding execution traces that are not relevant to the semantics of SMARTLTL.

[8]Calls to contract methods from the harness are also external calls.

```
1: procedure FEASIBLE(π)
2:     input: π = τ1 τ2^ω, lasso-shaped counterexample
3:     Output: ✓, ✗, or ?
4:     if ¬SAT(sp(τ1; τ2, true)) then return ✗
5:     V := {v | v ∈ Vars(τ2) ∧ t ∈ Txs(τ2) ∧ Read(t, v)} )
6:     V̄ := Vars(τ1; τ2)\V
7:     φ := ∃V̄.sp(τ1; τ2, true)
8:     if sp(τ2, φ) ⇒ φ then return ✓
9:     if FindRankingFn(τ1 τ2*) ≠ null then
10:        return ✗
11:    return ?
```

Algorithm 2: Feasibility checking procedure. Here, $\mathsf{sp}(s, \phi)$ denotes strongest postcondition of s wrt ϕ, and $\mathsf{Txs}(\tau_2)$ corresponds to contract transactions invoked in τ_2

Theorem VII.1. A smart contract $\mathcal{P}$ satisfies the specification (φ_F, φ_P) iff the corresponding Büchi contract $\mathcal{P} \times \mathcal{A}_{\varphi_F \wedge \neg\varphi_P}$ does not have a feasible infinite trace π such that $\pi \in \mathcal{L}(\mathcal{B})$.

D. Checking Feasability

We now describe how to check whether a path $\pi \in \mathcal{L}(\mathcal{B})$ is feasible. In general, paths that are accepted by the Büchi contract are lasso-shaped, meaning that π is always of the form $\tau_1\tau_2^\omega$. Intuitively, π is feasible if it is possible to execute τ_2 infinitely many times after executing the "stem" τ_1.

Our feasibility checking procedure is shown in Algorithm 2 and consists three conceptual steps, where $\mathcal{P}_{\tau_1\tau_2^\omega}$ represents the program $\tau_1;\ \mathsf{while(true)}\ \tau_2$:

1) Check feasibility of $\tau_1; \tau_2$ (line 4); if not, π is infeasible.
2) Check if $\mathcal{P}_{\tau_1\tau_2^\omega}$ is a non-terminating program (lines 5–8); if so, π is feasible.
3) Check if $\mathcal{P}_{\tau_1\tau_2^\omega}$ has a ranking function (lines 9). If so, $\mathcal{P}_{\tau_1\tau_2^\omega}$ is a terminating program; thus $\tau_1\tau_2^\omega$ is infeasible.

In this work, we use a simple but effective non-termination checking method that leverages domain knowledge for computing so-called *recurrent sets* [12]. In particular, suppose τ_2 involves contract methods F. Now, if the first execution of τ_2 is feasible and if the program state relevant to F never changes, then we know that subsequent executions of τ_2 will also be feasible. Essentially, lines 5–8 of Algorithm 2 implement this idea: here, ϕ captures program state relevant to τ_2 transactions after going through the loop once. If we can prove that executing τ_2 in a state satisfying ϕ always preserves ϕ, then this means we can execute τ_2 infinitely many times.

Theorem VII.2. If Algorithm 2 returns ✓ for a path π, then $\mathcal{P}_\pi$ is a non-terminating program.

Example VII.1. Consider the auction in Figure 2 and the infinite counterexample $\tau_1\tau_2^\omega$ given in Figure 15. This trace corresponds to a denial-of-service attack where one of the bidders prevents the other users from receiving their refund by reverting in its fallback method. To show this is a real counterexample, we need to prove that $\tau_1\tau_2^\omega$ is feasible — i.e., we can execute τ_2 infinitely many times after executing τ_1 once. Now, let ϕ be the strongest post-condition of feasible trace $\tau_1; \tau_2$. Clearly, if we have $\{\phi\}\tau_2\{\phi\}$, this means we can execute τ_2 infinitely many times.

However, in practice, if we compute ϕ as the strongest post-conditon of $\tau_1; \tau_2$, we can almost never prove the Hoare

τ_1 : $A.bid_{✓}(from: B_1, value: 15))$; ... ; $A.bid_{✓}(from: B_2, value: 16))$; ... ; $A.bid_{✓}(from: B_3, value: 17))$; ... ; $A.close_{✓}())$;
τ_2 : $A.refund_{✗}())$; ... ; $bidders_{✗}[0].transfer(refAmt)$; $revert()$;

Fig. 15: SMARTPULSE counterexample $\tau_1\tau_2^\omega$ from Section II-A, where A is the instance of the auction and B_i are the bidders

triple $\{\phi\}\tau_2\{\phi\}$ even when $\tau_1\tau_2^\omega$ is obviously feasible. This is because τ_2 involves blockchain-modifying statements in the harness in addition to invocations of the contract's transactions. However, the idea is that if the transactions called in τ_2 do not read from these modified blockchain states, then we can safely ignore those variables when determining feasibility of $\tau_1\tau_2^\omega$. Thus, in line 7 of Algorithm 2, we project out those variables through existential quantification.

Going back to our example, the only transaction involved in τ_2 is `refund`, which reads from variables `refAmt`, `bidders`, and `refunds`. Since these variables are not modified in τ_2, we have $\mathsf{sp}(\tau_2, \phi) \Rightarrow \phi$ at line 8 of Algorithm 2 because ϕ only involves these three variables. Thus, our method is able to prove the feasibility of $\tau_1\tau_2^\omega$.

VIII. IMPLEMENTATION AND LIMITATIONS

The implementation of SMARTPULSE consists of approximately 10,000 lines of code spanning three languages (C#, Java, C++) and supports most features of Solidity 0.5.x, including inheritance, hashing, and important built-in constructs such as send, transfer and call. Given a smart contract implemented in Solidity, SMARTPULSE first converts it to the Boogie intermediate representation [2] using a modified version of the VeriSol Solidity-to-Boogie translator [35]. Our modified version of VeriSol incorporates several optimizations, such as splitting VeriSol's memory maps using alias analysis to improve analysis scalability. The program transformations described in Section VI-A and Section VI-B are then performed on the Boogie code in preparation for verification, which also occurs at the Boogie level. Our implementation of the verification algorithm from Section VII is an extension of the UltimateAutomizer software model checker [8]. In particular, given a SMARTLTL specification, our implementation first constructs a Büchi contract as described in Section VII-C and tries to verify it using the CEGAR approach from Algorithm 1 and using the feasibility checking procedure discussed in Section VII-D. If the verifier discovers a counterexample, SMARTPULSE performs further post-processing and converts it to a program trace (i.e., sequence of transaction names and their arguments) that can be understood by users.

Modular verification. In addition to checking global temporal properties, SMARTPULSE can also be used to check method-level properties. That is, our implementation of SMARTPULSE also accepts SMARTLTL specifications at the method level.

Limitations. The current version of SMARTPULSE has several limitations. First, it does not support some low-level Solidity features, including inline assembly, singature-based calling mechanisms (e.g., `addr.call("function foo(int, int)")(a, b)`), bitwise operations, and Application Binary Interface (ABI) functions. Second, SMARTPULSE cannot reason precisely about non-linear arithmetic and models them using uninterpreted functions. Third, the current gas model used by SMARTPULSE is conservative and estimates gas usage from Solidity code rather than EVM bytecode.

In addition, the current version of SMARTPULSE makes several assumptions. For example, it assumes that miners are not adversarial and that transactions are executed in the order they are submitted. It also assumes that all attacks require only a single instance of the contract being analyzed. However, it is worth noting that the assumptions we make in this work are actually are less restrictive than prior work [17], [27]. In addition, SMARTPULSE can be easily extended to remove these assumptions, albeit potentially at the cost of scalability.

Finally, SMARTLTL only allows users to specify properties over contract variables at transaction boundaries. As a result, users cannot specify properties intended to query the behavior within a transaction. In addition, SMARTPULSE does not consider a property to be violated if the violation occurs within a transaction, but is not observable at the transaction boundary.

IX. EVALUATION

In this section, we present the results of our evaluation, which is designed to answer the following research questions: (1) Is SMARTPULSE able to verify liveness properties? (2) How does SMARTPULSE compare against existing smart contract verifiers? (3) Is SMARTPULSE able to generate attacks for vulnerable contracts?

With the exception of the results in Section IX-C, all experimental results reported in this section are conducted on a machine running MacOS 10.15.4 with an 8-Core Intel Core i9 and 16GB of memory. For these experiements, we also set a timeout of 5 hours and memory limit of 16GB. The results from Section IX-C were gathered on a machine running Ubuntu 18.04 with an Intel Xeon(R) W-3275 2.50 GHz CPU and 32GB of physical memory.

A. Liveness Evaluation

Since SMARTPULSE is the first automated approach for checking liveness properties of smart contracts, we first conduct an evaluation that focuses on liveness. To perform this evaluation, we consider smart contracts and properties considered in prior work on *interactively* verifying liveness properties [30].[9] In addition, we collect six other contracts (auctions, crowdsales, and escrows) that are deployed on the blockchain and that have interesting liveness properties. In particular, two of these contracts are taken from the popular OpenZeppelin library [25], and the remaining ones are taken from EtherScan (we selected ones where users deposited the most Ethereum throughout the contract's lifetime and that only use SMARTPULSE-supported features). For each of these contracts, we inspected their interface and formalized liveness properties we would *expect* them to satisfy. An English description of the properties for each benchmark can be found in Appendix D.

The results of this evaluation are summarized in Table I, which lists the contract's name, number of lines of code, and the number of properties that we checked. The column labeled "# verified" shows the number of properties that were successfully verified, and the "# refuted" column shows the

[9]Since the contracts considered in that work are written in Scilla, we manually translated them to Solidity.

Contract	LOC	# Properties	# Verified	# Falsified	Avg. Time (s)
RefundEscrow	129	2	2	0	333.2.0
EscrowVault	102	2	2	0	2587.6
RefundableCrowdsale	374	5	4	0	1124.8
EPXCrowdsale	171	5	5	0	442.2
Crowdfunding	55	3	3	0	19.8
ValidatorAuction	260	6	3	3	980.8
SimpleAuction	50	5	3	2	277.4
Auction	51	3	1	1	11.8
RockPaperScissors	66	3	3	0	8.5
Overall	1258	34	26	6	622.9

TABLE I: Liveness verification results

number of properties for which SMARTPULSE was able to provide a real counterexample. Finally, the column labeled "Time" shows the average analysis time in seconds.

The key take-away from Table I is that SMARTPULSE is able to successfully solve 32 of the 34 benchmarks (94%) but fails to terminate within the provided time limit for the remaining two. Out of 32 benchmarks on which the analysis terminates, SMARTPULSE reports counterexamples for six of them. In all six cases, we found that there was a discrepancy between the contract's implementation and the expected specification. For example, for the ValidatorAuction contract, one of the properties we specified is "losing bidders should be refunded the total sum of their bids". However, the actual implementation of this contract has two possible outcomes, namely `failed` and `DepositPending`. If the auction has failed (e.g., if there are not enough bidders), then all participants are issued their refund, so the expected property holds. On the other hand, if the auction has closed successfully, then the losing bidders are issued their refund *minus* some amount called `lowestSlotPrice`. In this case, if we change the specification to be consistent with the contract's actual behavior, then SMARTPULSE is able to verify the property.

Running time. The running time of SMARTPULSE varies from 7 seconds to 4833 seconds, with average running time being 623 seconds. On some benchmarks (e.g., EscrowVault), SMARTPULSE takes a very long time due to the large number of refinement steps that need to be performed. In general, the more complex the property, the longer SMARTPULSE takes to verify it. However, as we show in subsequent sections, SMARTPULSE is a lot faster when verifying safety properties.

B. Comparison with VERX on Temporal Safety Properties

Next, we compare SMARTPULSE against VERX, a state-of-the-art safety tool for verifying temporal *safety* properties of smart contracts. To perform this comparison, we first translated all VERX benchmarks from [27] to Solidity 0.5.0 and then wrote a corresponding SMARTLTL specification for each of the PastLTL specifications used by VERX. Since VERX performs verification under the *effective external callback freedom* assumption, we use SMARTPULSE's no re-entrancy attacker model when performing this evaluation.

The results of our VERX comparison are shown in Table II. Since VERX often needs the user to provide additional predicates to facilitate verification, we compare SMARTPULSE against two variants of VERX. For the first, labeled "VERX-USER", we supply VERX with the exact same set of predicates used in the VERX evaluation [27]. However, since SMARTPULSE is fully automated and does not require users to provide any predicates, this is not an apples-to-apples comparison. Thus, we also compare SMARTPULSE against a fully automated variant of VERX (labeled "VERX-AUTOMATED") in which we do not supply VERX with any predicates.

As shown in Table II, SMARTPULSE is able to verify significantly more benchmarks (43 vs. 81) compared to the fully automated version of VERX. When comparing SMARTPULSE against VERX-USER, SMARTPULSE still verifies more benchmarks within the 5 hour time limit, and the average running times of both tools are similar, with SMARTPULSE's median running time being faster.[10]

C. Large Scale Evaluation and KEVM-VER Comparison

In the previous subsections, we demonstrated the expressiveness and flexibility of SMARTPULSE by using it to verify several liveness and temporal safety properties. However, because this evaluation requires writing custom specifications for each contract, we were only able to evaluate SMARTPULSE on 22 contracts. In this section, we perform a larger scale evaluation on ERC20 contracts that all share the same specification. We chose ERC20 contracts because they are the most widely used contracts on the Ethereum blockchain, making up 72.9% of all high-activity contracts [24]. While the correctness properties of ERC20 contracts are much simpler than those considered in Sections IX-A and IX-B, this evaluation allows us to demonstrate that SMARTPULSE can be used to analyze a large number of contracts. Furthermore, since a prior research effort [26], henceforth called KEVM-VER, has also been evaluated on ERC20 contracts, this evaluation allows us to perform a comparison against an additional state-of-the-art verification tool for smart contracts.

To perform this large scale evaluation, we collected from EtherScan the 200 (unique) most widely-used ERC20 contracts whose source is available. Among these 200 contracts, 49 of them cannot be analyzed by KEVM-VER, which leaves us with a total of 151 contracts to evaluate on. In terms of correctness specifications, we utilize the ERC20-K formalism proposed in prior work [28] that serves as a complete formal specification of the ERC20 standard in the K framework [29]. However, since SMARTPULSE does not consume K specifications, we translated these specifications to equivalent (but often much shorter) SMARTLTL properties. Furthermore, since ERC20-K specifications are in the form of method pre- and post-conditions, we also use method-level specifications, but expressed in SMARTLTL.

The results of this evaluation are shown in Table III. Here, rows correspond to each of the 12 ERC20 specifications, and the right and left halves of the table provide statistics about SMARTPULSE and KEVM-VER respectively. In particular, for each tool, we report (1) the number of ERC20 contracts that could be verified, (2) the number of contracts that were proven to violate the property, (3) the number of contracts for which

[10] At the time of writing, VERX was not freely available. However, the authors of VERX provided us with server access that allowed us to run these experiments. Since the two tools are run on different machines, we note that the reported running times do not provide an apples-to-apples comparison.

Project	# Properties	SmartPulse # Verified	SmartPulse Average Time (s)	SmartPulse Median Time (s)	VerX-User # Verified	VerX-User Average Time (s)	VerX-User Median Time (s)	VerX-Automated # Verified	VerX-Automated Average Time (s)	VerX-Automated Median Time (s)
Overview	4	4	88.7	80.6	4	116.9	89.2	0	——	——
Alchemist	3	2	3.8	3.8	3	40.7	40.7	2	41.6	41.6
Brickblock	6	6	17.9	19.2	6	110.2	112.4	3	53.1	43.3
Crowdsale	9	9	91.5	31.4	9	95.3	94.4	0	——	——
ERC20	9	9	396.9	9.8	9	25.8	21.6	9	25.8	21.6
ICO	8	7	29.6	21.7	8	1749.4	1697.2	0	——	——
Mana	4	4	222.8	152.2	0	——	——	0	——	——
Melon	16	16	821.4	12.0	16	112.0	62.9	16	28.9	27.6
MRV	5	5	32.7	24.1	5	393.5	350.1	5	426.8	350.1
PolicyPal	4	4	85.2	27.1	0	——	——	0	——	——
VUToken	5	5	138.9	75.3	4	445.9	216.9	0	——	——
Zebi	5	5	93.5	22.8	5	178.8	27.4	3	17.5	13.3
Zilliqa	5	5	1168.3	7.9	5	40.2	31.1	5	40.2	31.1
Overall	83	81	328.6	20.1	74	310.4	69.0	43	77.3	29.5

TABLE II: Comparison against VerX on temporal safety properties.

Property	SmartPulse # Verified	SmartPulse # Falsified	SmartPulse # Unknown	SmartPulse Average Time (s)	KEVM-Ver # Verified	KEVM-Ver # Falsified	KEVM-Ver # Unknown	KEVM-Ver Average Time (s)
TotalSupply	108	35	7	2.9	89	37	25	390.1
BalanceOf	133	10	7	2.7	125	1	25	391.9
Allowance	139	4	7	2.6	123	3	25	397.2
Approve	69	74	7	4.1	62	64	25	402.4
Transfer-Normal	41	101	8	4.4	42	80	29	403.6
Transfer-Self	74	69	7	4.3	75	47	29	400.8
Transfer-Fail	124	20	7	3.5	106	17	28	404.3
Transfer-Self-Fail	138	6	7	2.8	123	3	25	405.9
TransferFrom-Normal	38	101	11	19.0	39	83	29	416.8
TransferFrom-Self	38	103	9	8.6	38	84	29	412.4
TransferFrom-Fail	124	19	8	4.6	105	18	28	414.5
TransferFrom-Self-Fail	131	13	7	3.3	114	9	28	405.1
Overall	1164	556	92	5.2	1041	446	325	403.7

TABLE III: Large Scale Evaluation

the tool did not report a result (e.g., due to a time-out), and (4) average time in seconds for those benchmarks that could be solved (i.e., either verified or refuted).

The key take-away from this evaluation is that SmartPulse is able to solve significantly more benchmarks than KEVM-Ver (95% vs 82%), and it is able to solve them a lot faster (5 vs 404 seconds). We conjecture that SmartPulse is significantly faster than KEVM-Ver due to its use of lazy abstraction as opposed to eager symbolic execution.

To ensure the correctness of these results, we also compared the results produced by SmartPulse against those of KEVM-Ver. For the benchmarks solved by both tools, SmartPulse and KEVM-Ver produced the same results except in two cases. Upon further inspection, we found these two discrepancies to be caused by a bug in KEVM-Ver (specifically, a bug in the translation from EVM bytecode to K), and we manually confirmed that the result produced by SmartPulse is indeed correct.

D. *Evaluating* SmartPulse *on Vulnerable Contracts*

One of the capabilities provided by SmartPulse is the ability to generate attacks for vulnerable contracts. In this section, we evaluate SmartPulse on a set of benchmarks that contain vulnerability patterns described in prior work [11], [17], [22], [23] and assess whether SmartPulse can generate attacks for all of these vulnerable contracts. Since several of these vulnerability patterns require non-trivial fallback implementations to perform the attack, we conduct this evaluation using the powerful adversary model.

The results of this evaluation are shown in Table IV. Here, the column labeled "Pattern" describes the vulnerability patterns (e.g., re-entrancy, integer overflow) described in prior work, and the column labeled "Property" shows an important correctness property that is violated due to the presence of the corresponding vulnerability pattern. As shown in Table IV, SmartPulse is able to generate attacks for all of these benchmarks, and the column labeled "Attack summary" shows a summary of the attack in terms of relevant methods that are invoked. We highlight some of the salient features of the attacks generated by SmartPulse:

Counterexamples as attacks. The counterexamples generated by SmartPulse provide a series of transactions along with their argument values that, if executed, would violate the given temporal property. Therefore, the counterexamples generated by SmartPulse correspond to a full attack against the vulnerable contract (e.g., see Figure 15).

Synthesis of attacker's fallback. As indicated by $\rightsquigarrow^f$ the notation in Table IV, about a third of the benchmarks require synthesizing a fallback method for an adversarial contract. In the process of generating a counterexample trace, SmartPulse also synthesizes a concrete implementation of the attacker's fallback method.

Infinite counterexample traces. Denial-of-service vulnerabilities correspond to violations of liveness properties; hence, for the DOS vulnerability patterns, SmartPulse generates an attack in the form of an infinite counterexample trace. For instance, for revert DOS, the attack involves a loop of the form $(\mathsf{func} \rightsquigarrow \mathsf{transfer} \rightsquigarrow^f \mathsf{revert})^\omega$, indicating that all calls to function func result in the attacker reverting that transaction.

Attacks for gas vulnerabilities. To the best of our knowledge, SmartPulse is the first tool that can generate attacks for gas-related vulnerabilities. For instance, in the attack for the "gas DOS" pattern, SmartPulse's attack involves creating enough bidders so that the contract runs out of gas when trying to issue refunds.

Pattern	Property	Attack Summary
Reentrancy	Users can't withdraw more money than their credit	deposit $\rightarrow$ deposit $\rightarrow$ withdraw $\rightarrow$ call $\rightarrow^{f}$ withdraw
Unprotected Function	Users can't withdraw unless they are the owner	addOwner $\rightarrow$ withdraw
Integer Overflow	A balance cannot decrease without withdrawing	deposit $\rightarrow$ deposit
Integer Underflow	A balance cannot be greater than the sum of deposits	transfer
Gas DOS	Users who buy tokens are eventually refunded	buy $\rightarrow$ buy $\rightarrow$ close $\rightarrow$ (refund $\rightarrow$ transfer $\rightarrow$ transfer $\rightarrow$ OOG)$^{\omega}$
Revert DOS	Users can always eventually outbid another user	bid $\rightarrow$ (bid $\rightarrow$ transfer $\rightarrow^{f}$ revert)$^{\omega}$
Push DOS	All non-winning bidders are eventually refunded	bid $\rightarrow$ bid $\rightarrow$ bid $\rightarrow$ close $\rightarrow$ (refund $\rightarrow$ transfer $\rightarrow^{f}$ revert)$^{\omega}$
Unchecked Send	The contract tracks how much money each user stores	deposit $\rightarrow$ withdraw $\rightarrow$ send $\rightarrow$ revert
Locked Funds	Funds cannot be transferred to unaccessible accounts	deposit $\rightarrow$ transfer

TABLE IV: Evaluation for attack generation. Here $\rightarrow^{f}$ indicates a call from a fallback function and OOG stands for "Out of Gas."

X. RELATED WORK

A. Verification of Smart Contracts

In recent years, there has been great interest in formally verifying the correctness of smart contracts. For instance, ZEUS [17], VERISOL [34], SOLC-VERIFY [13], and KEVM-VER [26] allow users to specify correctness properties in terms of method pre- and post-conditions and verify the program by generating verification conditions and discharging them with an SMT solver. Since VERISOL and SOLC-VERIFY do not automate invariant generation, they require users to manually provide annotations. Zeus, on the other hand, can generate invariants using a Constrained Horn Clause (CHC) solver, and KEVM-VER translates EVM bytecode to KEVM and leverages the K framework [29] for verification. However, all of the techniques are limited to safety and require the user to write lower-level specifications compared to SMARTPULSE.

Among verification techniques for smart contracts, our method is most closely related to VERX [27], which performs semi-automated verification of temporal safety specifications written in PastLTL. VERX performs a combination of symbolic execution and predicate abstraction; furthermore, since VERX extracts predicates automatically from the contract's source code, it is capable of automated verification. However, VERX does not perform abstraction refinement; thus, the counterexamples it produces can be spurious, and successful verification may require the user to supply additional predicates. SMARTPULSE differs from VERX in that it is not limited to safety and never produces spurious counterexamples.

The only prior work that addresses liveness properties of smart contracts is by Sergey et al [31]. They express smart contracts in an intermediate language called SCILLA [31] and manually discharge proofs using the Coq proof assistant [7]. In contrast, our method is fully automated and can perform falsification as well as verification.

B. Finding Bugs in Smart Contracts

There has also been significant interest in characterizing and detecting vulnerability patterns in smart contracts. For instance, the Oyente tool by Luu et al. uses symbolic execution to check for various vulnerability patterns such as reentrancy [19]. Similarly, MADMAX [11] uses dataflow analysis to check for out-of-gas related vulnerabilities, and Feist et al. [9] describe the Slither infrastructure for building scalable static bug finding tools. In contrast to our method, these tools cannot be used to verify functional correctness. Furthermore, because patterns like re-entrancy do not always lead to the violation of a correctness property, these approaches can erroneously flag safe contracts as being vulnerable.

C. Modeling Exceptions in Smart Contracts

One of the challenges we addressed is how to faithfully model revert statements when reasoning about LTL properties. Among prior techniques, KEVM [15] and Lem [16] model reverts by restoring the initial state of the reverting call or transaction. On the other hand, VERISOL [35] and SOLC-VERIFY [13] mark any paths that throw an exception as infeasible. Such modeling, however, is unsound in the presence of low-level calls (e.g., `send`) since these methods do not revert a transaction if one of their callees throws an exception. In contrast to these approaches, our method instruments Solidity programs with history variables to capture the. Our approach is amenable to automated verification since it explicitly marks which paths modify the contract's state and which revert.

D. Verification using CEGAR

Our verification algorithm is based on the counterexample-guided abstraction refinement paradigm [4], [5]. The general idea is to perform verification using a coarse abstraction and then iteratively refine it as spurious counterexamples are encountered. Most CEGAR techniques generalize counterexamples by using Craig interpolation [14], [21], with the goal of ruling out more than a single counterexample. However, in general, CEGAR-based software model checkers do not have termination guarantees, and our method inherits this limitation.

E. Checking Liveness Properties

A number of approaches have been proposed to verify liveness properties of infinite-state systems. Several approaches verify liveness by searching for a program path that violates the LTL property [3], [6], [8], [33]. They do so by reducing liveness verification to fair termination, then search for a path that does not fairly terminate using a combination of SMT solvers and ranking function synthesizers. Our verification approach is based on the same framework proposed by Dietsch et al. [8] but differs in two important ways: First, due to the semantics of SMARTLTL, our product construction only needs to consider external call/return sites as final states. Second, for non-termination checking, we use a simple but effective technique that leverages the distinction between variables used in the harness vs. those used in the transactions themselves. In other words, rather than using expensive techniques for computing recurrent sets [12], we can check non-termination in a much simpler way. As we show in Appendix C, these differences are very important for making liveness verification practical in this context.

XI. CONCLUSION AND FUTURE WORK

We have described SMARTPULSE, the first tool for automatically checking general temporal properties of smart contracts, including liveness. Given a formal SMARTLTL specification and an attacker model, SMARTPULSE first performs a sequence of program instrumentations to model the contract's execution environment and then uses a CEGAR-based verification approach to search for property violations. We evaluate SMARTPULSE on a total of 1947 benchmarks and

demonstrate that SMARTPULSE advances the state-of-the-art in smart contract verification.

There are several interesting avenues for future work. First, we are interested in exploring new, more scalable algorithms for finding non-terminating program traces. Second, we are interested in proving the end-to-end soundness of our tool. However, since Solidity does not have formal semantics, the proof would need to be with respect to an intermediate representation like YUL that does have formal semantics.

REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: temporal property verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–393. Springer, 2016.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[6] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 265–276. ACM, 2007.

[7] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[8] D. Dietsch, M. Heizmann, V. Langenfeld, and A. Podelski. Fairness modulo theory: A new approach to ltl software model checking. In *International Conference on Computer Aided Verification*, pages 49–66. Springer, 2015.

[9] J. Feist, G. Greico, and A. Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15. IEEE Press, 2019.

[10] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In *International Conference on Computer Aided Verification*, pages 53–65. Springer, 2001.

[11] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[12] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. *SIGPLAN Not.*, 43(1):147–158, Jan. 2008.

[13] Á. Hajdu and D. Jovanovic. solc-verify: A modular verifier for solidity smart contracts. In S. Chakraborty and J. A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2019.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.

[15] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217. IEEE Computer Society, 2018.

[16] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[17] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, pages 1–12, 2018.

[18] B. C. P. Ltd. Parity multisig recovery reconciliation. https://github.com/bokkypoobah/ParityMultisigRecoveryReconciliation.

[19] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[20] Z. Manna and A. Pnueli. *Temporal verification of reactive systems*. Springer Science & Business Media, 2012.

[21] K. L. McMillan. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*, pages 123–136. Springer, 2006.

[22] MythX. Smart contract weakness classification registry. https://github.com/SmartContractSecurity/SWC-registry.

[23] T. of Bits. (not so) smart contracts. https://github.com/crytic/not-so-smart-contracts.

[24] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, pages 1–41, 2020.

[25] OpenZeppelin. Openzeppelin: Build secure smart contracts in solidity. https://openzeppelin.com/contracts/.

[26] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915, 2018.

[27] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.

[28] G. Rosu. December 2017. erc20-k: Formal executable specification of erc20.

[29] G. Roșu and T. F. Șerbănuță. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[30] I. Sergey, A. Kumar, and A. Hobor. Temporal properties of smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 323–338. Springer, 2018.

[31] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019.

[32] D. Siegel. Understanding the dao attack. *CoinDesk*. https://www.coindesk.com/understanding-dao-hack-journalists.

[33] M. Wang, C. Tian, N. Zhang, and Z. Duan. Verifying full regular temporal properties of programs via dynamic program execution. *IEEE Transactions on Reliability*, 68(3):1101–1116, 2018.

[34] Y. Wang, S. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers*, page 87. Springer Nature, 2019.

[35] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer. Formal specification and verification of smart contracts for azure blockchain, 2018.

APPENDIX

A. SMARTLTL *Formal Semantics*

We formalize the semantics of SMARTLTL specifications in terms of *execution traces* of smart contracts. For the purposes of this paper, an execution trace τ is a sequence of triples of the form (f, κ, σ) where f is the name of an external function, $\kappa \in \{\mathsf{call}(\overline{x}), \mathsf{return}, \mathsf{revert}\}$, and σ is a *valuation* mapping SMARTLTL terms to their values. In the remainder of this section, we make the following assumptions about a trace: First, free variables X occurring in SMARTLTL specification are initialized to a non-deterministic value in σ. Second, we assume that the trace is pre-processed so that any external call that occurs within a reverting transaction has a corresponding revert rather than return.

To facilitate our formalization, we define some useful operations over execution traces. First, given a trace τ, we write $\hat{\tau}$ to denote the last item in the sequence. Next, given an index or trace element α, we write $\mathsf{Pre}_\tau(\alpha)$ (resp. $\mathsf{Post}_\tau(\alpha)$) to denote the prefix of τ up to and including (resp. after and not including) α. If a trace element α corresponds to a call, we write $\mathsf{Success}_\tau(\alpha)$ to indicate that α has a corresponding return element. Finally, for trace elements that correspond to

$$[\![e]\!]_\tau = \sigma(e) \text{ where } e \text{ is a free variable or Solidity expression and } \hat{\tau} = (_, _, \sigma)$$
$$[\![\mathsf{old}(v)]\!]_\tau = \sigma(v) \text{ where } \hat{\tau} = (f, \mathsf{call}(\overline{x}), \sigma)$$
$$[\![\mathsf{old}(v)]\!]_\tau = \sigma'(v) \text{ where } \hat{\tau} = (f, \mathsf{return/revert}, \sigma),\ \mathsf{Call}_\tau(\hat{\tau}) = (f, \mathsf{call}(\overline{x}), \sigma')$$
$$[\![\mathsf{csum}(v)]\!]_\tau = \sum_{k \in \mathsf{keys}(\sigma(v))} \sigma(v[k]) \text{ where } \hat{\tau} = (_, _, \sigma)$$
$$[\![\mathsf{fsum}(f, v, \psi)]\!]_\tau = \sum_{\sigma \in R} \sigma(v) \text{ where } R = \{\sigma \mid (\alpha = (f, \mathsf{call}(\overline{x}), \sigma)) \in \tau,\ \mathsf{Success}_\tau(\alpha),\ \tau' = \mathsf{Pre}_\tau(\alpha),\ [\![\psi]\!]_{\tau'} \equiv \mathsf{True}\}$$

Fig. 16: Evaluation of SMARTLTL terms and predicates. We omit standard boolean/arithmetic operators.

$$\frac{\mathcal{F}_0 = (f, \mathsf{call}(\overline{x}), \sigma) \quad [\![\psi]\!]_{\mathcal{H}:\mathcal{F}_0} \equiv \mathsf{True}}{\mathcal{H}, \mathcal{F} \vDash \mathsf{start}(f, \psi)}$$

$$\frac{\mathcal{F}_0 = (f, \mathsf{return}, \sigma) \quad [\![\psi]\!]_{\mathcal{H}:\mathcal{F}_0} \equiv \mathsf{True}}{\mathcal{H}, \mathcal{F} \vDash \mathsf{finish}(f, \psi)}$$

$$\frac{\mathcal{F}_0 = (f, \mathsf{revert}, \sigma) \quad \alpha = \mathsf{Call}_{\mathcal{H}:\mathcal{F}_0}(\mathcal{F}_0) \quad [\![\psi]\!]_{\mathcal{H}:\mathsf{Pre}_{\mathcal{F}}(\alpha)} \equiv \mathsf{True}}{\mathcal{H}, \mathcal{F} \vDash \mathsf{revert}(f, \psi)}$$

$$\frac{\mathcal{F} = \alpha : \tau \quad \mathcal{H} : \alpha,\ \tau \vDash \varphi}{\mathcal{H}, \mathcal{F} \vDash \circ\varphi}$$

$$\frac{(\mathcal{H} : \mathsf{Pre}_{\mathcal{F}}(k)), \mathsf{Post}_{\mathcal{F}}(k) \vDash \varphi_2 \quad \forall i < k.\ (\mathcal{H} : \mathsf{Pre}_{\mathcal{F}}(i)), \mathsf{Post}_{\mathcal{F}}(i) \vDash \varphi_1}{\mathcal{H}, \mathcal{F} \vDash \varphi_1\ \mathcal{U}\ \varphi_2}$$

$$\frac{[], \tau \vDash \varphi}{\tau \vDash \varphi}$$

Fig. 17: Semantics of SMARTLTL formulas

either returns or reverts, we use the notation $\mathsf{Call}_\tau(\alpha)$ to denote the corresponding call event to f.

Since SMARTLTL expressions allow aggregating values over the entire history of the contract execution (e.g., recall the fsum construct), SMARTLTL expressions and predicates are evaluated over execution trace prefixes rather than valuations. Specifically, given a SMARTLTL expression e and a trace prefix τ, we write $[\![e]\!]_\tau$ to denote the result of evaluating e as defined in Figure 16.

Figure 17 presents the semantics of formulas in our specification language. Given a formula φ and a trace τ, we say that τ is a *model of* φ, written $\tau \models \varphi$, if φ evaluates to true under τ. Our entailment relation $\models$ for SMARTLTL formulas is given in Figure 17. Since the temporal operators $\circ$ and $\mathcal{U}$ are functionally complete, Figure 17 omits the semantics of the remaining temporal operators, which can be desugared using known techniques into next and until operators [20].

Unlike standard LTL where the semantics of a formula is defined in terms of a single trace, Figure 17 uses an auxiliary judgment of the form $\mathcal{H}, \mathcal{F} \vDash \varphi$ that utilizes a *pair* of traces $(\mathcal{H}, \mathcal{F})$ where $\mathcal{H}$ represents the "history" (excluding the present) and $\mathcal{F}$ represents the "future" (including the present). We define our semantics in this manner because the evaluation of SMARTLTL expressions and predicates requires having access to the entire execution history up until the current time.

Given a full specification $\varsigma \equiv (\varphi_\mathsf{F}, \varphi_\mathsf{P})$ consisting of fairness assumptions φ_F and property φ_P, we say that an execution trace τ satisfies ς, denoted $\tau \models \varsigma$, if and only if (a) either $\tau \not\models \varphi_\mathsf{F}$, or (b) $\tau \models \varphi_\mathsf{P}$. Finally, given a Solidity contract $\mathcal{P}$ and SMARTLTL specification ς, we say that $\mathcal{P}$ satisfies the specification, written $\mathcal{P} \models \varsigma$, if, for all feasible execution traces τ of $\mathcal{P}$, we have $\tau \models \varsigma$.

B. Conformance Checking Theorems

Theorem A.1. A smart contract $\mathcal{P}$ satisfies the specification $(\varphi_\mathsf{F}, \varphi_\mathsf{P})$ iff the corresponding Büchi contract $\mathcal{P} \times \mathcal{A}_{\varphi_\mathsf{F} \wedge \neg\varphi_\mathsf{P}}$ does not have a feasible infinite trace π such that $\pi \in \mathcal{L}(\mathcal{B})$.

Proof. We provide a sketch of the proof here. First, we use the construction of the Büchi Contract to prove that there is a feasible trace $\pi = (s_1, assume(a_1)), (s_2, assume(a_2)), ... \in Traces(\mathcal{B})$ iff there is a feasible trace $\pi_P = s_1, s_2, ... \in Trace(\mathcal{P})$ and $\pi_\mathcal{A} = a_1, a_2, ... \in Traces(\mathcal{A})$ such that $\pi_P \models \pi_\mathcal{A}$. In addition, we establish the construction of $\mathcal{A}$ from $\varphi_\mathsf{F} \wedge \neg\varphi_\mathsf{P}$ is correct from prior work [10].

With this, we can prove that $\mathcal{P}$ satisfies the specification if $\mathcal{L}(\mathcal{B})$ doesn't have an infinite path. If this were not the case, then there must be a feasible infinite path $\pi \in \mathcal{L}(\mathcal{B})$ but $\mathcal{P}$ satisfies the specification. We know that there must also be feasible paths $\pi_P \in Trace(\mathcal{P})$ and $\pi_A \in Traces(\mathcal{A})$ such that $\pi_P \models \pi_A$. If $\pi \in \mathcal{L}(\mathcal{B})$ then so to must $\pi_A \in \mathcal{L}(\mathcal{A})$ from the construction of the Büchi Contract. There is therefore a contradiction since a feasible infinite path $\pi_P \in Trace(\mathcal{P})$ has a corresponding $\pi_A \in \mathcal{L}(\mathcal{A})$ such that $\pi_P \models \pi_A$.

Now assume that $\mathcal{L}(\mathcal{B})$ does not have a feasible infinite trace $\pi \in \mathcal{L}(\mathcal{B})$ and $\mathcal{P}$ does not satisfy the specification. There must therefore exist a feasible infinite path $\pi_P \in Traces(P)$ and $\pi_A \in \mathcal{L}(\mathcal{A})$ such that $\pi_P \models \pi_A$. In addition, we also know that there must be a path $\pi \in Traces(\mathcal{B})$, but $\pi \notin \mathcal{L}(\mathcal{B})$. From the construction of the Büchi Contract, this must be due to one of two cases. (1) An external call or return does not occur infinitely often. For this to occur, π_P must spend infinite time inside a transaction, which violates the Solidity semantics. Thus π_P must not be feasible. (2) All accepting states visited infinitely often in π_A must be outside of an external call or return. From the semantics of SMARTLTL we know that atoms are only checked at external call boundaries. Thus, π_P and π_A do not violate the specification according to SMARTLTL. □

Theorem A.2. If Algorithm 2 returns ✓ for a path π, then $\mathcal{P}_\pi$ is a non-terminating program.

Proof. Consider a path $\pi = \tau_1\tau_2^\omega$ that is not feasible but Algorithm 2 returns ✓. For this to occur $sp(\tau_2, \phi) \Rightarrow \phi$, where $\phi = \exists\overline{V}.sp(\tau_1; \tau_2, \top)$, must not indicate that a path is infinite. When $\overline{V} = \emptyset$, however, $sp(\tau_2, \phi) \Rightarrow \phi$ is equivalent to checking the Hoare triple $\{\phi\}\ \tau_2\ \{\phi\}$ which does prove that $\tau_1\tau_2^\omega$ is infinite. Thus the definition of $\overline{V}$ must be incorrect. For this to occur, there must be a variable $v \in \overline{V}$ that is relevant to the feasibility of $\tau_1\tau_2^\omega$. From the construction of $\overline{V}$ on lines 5–6 of Algorithm 2, v must also not be read by any of the transactions in τ_2. It is therefore the case that v does not introduce any dependencies between subsequent executions of transactions from τ_2. Since we want to determine if the transactions in τ_2 can be executed infinitely often, v must

Benchmark	SmartPulse Vulnerable Time (s)	SmartPulse Fixed Time (s)	SmartPulse-NoK Vulnerable Time (s)	SmartPulse-NoK Fixed Time (s)
Reentrancy	17.5	80.2	Unknown	Unknown
Unprotected Fn.	11.4	15.6	Unknown	15.2
Int Overflow	5.9	2.3	Unknown	2.3
Int Underflow	10.6	88.5	Unknown	81.8
Gas DOS	842.5	51.0	Unknown	50.8
Revert DOS	9.3	4.9	Unknown	5.2
Push DOS	277.1	317.9	Unknown	371.2
Unchecked Send	9.1	4.2	Unknown	4.1
Locked Funds	17.2	15.2	Unknown	14.4

TABLE V: Ablation study of SmartPulse on attack generation benchmarks. Here, Unknown indicates that the verifier returned Unknown as its result.

therefore be irrelevant. Since v cannot be both relevant and irrelevant, there is a contradiction. □

C. Ablation Study

In this section we evaluate the benefits of adding domain knowledge to the verification process. To do so, we created a version of SmartPulse without any of the additions described in Section VII, called SmartPulse-NoK. We then evaluated these two tools on the set of attack generation benchmarks from Section IX-D with the powerful adversary model.

The results of the evaluation are given in Table V. Here, we consider two variants of each benchmarks: (1) the vulnerable version, and (2) the fixed version that satisfies the corresponding property. The columns named "Vulnerable Time" (resp. "Fixed Time") give the amount of time required for the given tool to find a violation of (resp. to verify) the property for the vulnerable (resp. fixed) variant of each benchmark. These results show that SmartPulse-NoK is *not* able to find the violation in any of the vulnerable contracts and verifies fewer of the fixed contracts than SmartPulse. In addition, for the contracts verified by both tools, SmartPulse and SmartPulse-NoK have similar runtimes.

In all cases, SmartPulse-NoK is unable to find a violation of the property in the vulnerable contract. For all but the Reentrancy violation, SmartPulse-NoK considers the same feasible path $\tau_1\tau_2^\omega$ as SmartPulse, however it fails to prove its feasibility. This result highlights the usefulness of our non-termination checker when proving feasibility (see Section VII-D).

For the Reentrancy pattern, SmartPulse-NoK is unable to find a violation in the vulnerable contract, and it is also unable to verify the fixed contract. In both cases, the problem is that SmartPulse-NoK finds a spurious path $\tau_1\tau_2^\omega$ but fails to prove it is infeasible – i.e., it fails to prove termination. Even though SmartPulse also uses the same method for finding ranking functions, it does not need to prove the infeasibility of the same path $\tau_1\tau_2^\omega$ due to the way the Büchi contract is constructed.

D. Liveness Properties

This section contains the properties verified in Section IX-A, which are shown in Table VI. Here the column labeled "Contract" identifies the contract the property is verified against and column labeled "Property" gives an English description of the property. Each contract and their associated properties labeled with $\star$ was taken from prior work and the remaining properties were written by us. In Section IX-A, all Crowdfunding properties are also evaluated against RefundableCrowdsale and EPXCrowdsale. In addition, all Auction properties are also evaluated against ValidatorAuction and SimpleAuction.

Contract	Property
RefundEscrow	If a user withdraws funds after refunds are enabled, they will eventually be sent the sum of their deposits.
	If the beneficiary withdraws after the escrow is ended, they will eventually be sent the sum of all deposits.
EscrowVault	If a user requests a refund after refunds are enabled, they will eventually be sent the sum of their deposits.
	If the escrow is closed, the beneficiary will only be sent the sum of all deposits.
Crowdfunding⋆	The contract's accounted funds do not decrease unless the campaign has been funded or the deadline has expired.
	The contract preserves records of individual donations by backers, unless they interact with it.
	If the campaign fails, the backers can eventually get their refund.
RefundableCrowdsale	A user who claims a refund after the crowdsale is finalized will eventually be sent the sum they spent on tokens.
	If the crowdsale is closed and the goal has been reached, the beneficiary is eventually sent all funds used to buy tokens.
EPXCrowdsale	If a user issues a refund, they are eventually sent the sum they spent on tokens
	If funds are released to the beneficiary and they attempt to claim them, they will be sent the all funds used to buy tokens.
Auction⋆	The balance should be greater than or equal to the sum of the highest bid and all pending returns.
	For some account a, the contract should track the sum of all transfers a has made with the contract.
	Anyone other than the highest bidder should be able to retrieve the full amount of their bids from the contract exactly once.
ValidatorAuction	If the auction is closed and a user requests a withdraw, they will eventually be sent the sum of their bids.
	If a user withdraws in the $DepositPending$ state, they will eventually be sent the sum of their bids minus the lowest price.
	If a user attempts to withdraw in the state $Failed$, they will eventually be sent the sum of their bids.
SimpleAuction	If a user's bid is outbid, they will eventually be sent back at least the value of that bid.
	If the auction is ended and at least one bid was made, the beneficiary is sent the value of the highest bid.
RockPaperScissors⋆	No other party besides player 1, player 2 or the owner can be awarded the prize, which is equal to the contract's balance.
	Player 1 can only submit a valid choice once and the contract will not accept an invalid choice.
	Player 2 can only submit a valid choice once and the contract will not accept an invalid choice.

TABLE VI: Liveness Properties. Note, all Crowdfunding properties are also evaluated against RefundableCrowdsale and EPXCrowdsale, and all Auction properties are also evaluated against ValidatorAuction and SimpleAuction.