Tainting in Smart Contracts: Combining Static and Runtime Verification*

Shaun Azzopardi[2][0000−0002−2165−3698], Joshua Ellul[1][0000−0002−4796−5665], Ryan Falzon[3][0000−0001−7513−3658], and Gordon J. Pace[1][0000−0003−0743−6272]

[1] University of Malta, Msida, Malta
joshua.ellul@um.edu.mt and gordon.pace@um.edu.mt
[2] University of Gothenburg, Gothenburg, Sweden shaun.azzopardi@gu.se
[3] Hash Data ryan@hashdata.co

Abstract. Smart contracts exist immutably on blockchains, making their pre-deployment correctness essential. Moreover, they exist openly on blockchains — open for interaction with any other smart contract and offchain entity. Interaction, for instance with off-chain oracles, can affect the state of the smart contract, and correctness of these smart contracts may depend on the trustworthiness of the data they manipulate or events they generate which, in turn, would depend on which parties or what information contributed to them.
In this paper, we develop and present dynamic taint analysis techniques to enable data tainting in smart contracts. We propose an extension of Solidity that enables labelling inputs of interaction endpoints with dynamic data-carrying labels that capture actionable information about the sender. These labels can then be propagated dynamically across transactions to transitively dependent data. Specifications can then refer to such taints, for instance for ensuring that certain data could not have been influenced through interaction by a certain party. We further allow the use of taints as part of the language, affecting the control flow of the smart contract. To manage the overheads of such runtime tainting we develop sound static analysis-based techniques to prune away unnecessary instrumentation. We give a case study as a proof-of-concept, and measure the overheads associated with our additions before and after optimisation.

Keywords: taint analysis · runtime verification · static analysis.

1 Introduction

Smart contracts on blockchains are programs that promise dependability through immutability and code transparency. However, this is not enough to ensure cor-

* This research has received funding from the ERC consolidator grant D-SynMA (No. 772459), the University of Malta Research Awards project *"Systematising Smart Contracts within Classical Contract Law Theory"*, and the European Agricultural Fund for Rural Development project *"VinoVeritas: An Authority to Consumer Wine Audit Solution"*.

rectness of the smart contracts. Formal methods have been applied for this purpose, to allow for some level of security and verification of functional properties of smart contracts, e.g., [7, 5, 1]. One interesting aspect of smart contracts is the ability of smart contracts to interact with each other or with off-chain entities. This interaction is the only way in which smart contracts can change state, with each (data-carrying) interaction forming part of a *transaction*. While the blockchain on which smart contracts are executed is decentralised in nature, the logic of a smart contract or data upon which it depends may not be. Consider, for instance, a betting smart contract depending on random numbers provided by third party oracles, or an insurance smart contract depending on reports by experts and information provided by a user. Whenever a smart contract's domain extends beyond what is digital and resides on the blockchain, it must interact with the real-world which is, by its very nature, centralised. A temperature sensor is, for instance, such a centralised point-of-trust, and, unless one goes to great lengths to have multiple independent sensors, the readings it provides and any logic or data which depend on them should ideally be tagged as such.

Given an event of interest (e.g., upgrading the level of a user or the violation of a property) it is interesting to reason about the contributing causes to this event, including any contributing interactions. However, such information is not typically available given interactions may be separated far in time from the events of interest to which they contribute. We observe that this kind of reasoning has been explored in literature, to an extent, in the study of *taint analysis* [12].

Taint analysis is typically concerned with identifying when input to a program can pose a security risk, e.g., if it can cause dangerous commands to be executed. Untrustworthy input is said to be *tainted*, while the sensitive parts of the program are called *sinks*, and the problem then is to find out whether tainted data can enter sinks. There are two approaches to taint analysis: static or dynamic. Identifying problems statically, pre-deployment, is ideal but having a sound and complete analysis is, in general, impossible forcing one to resort to over- or under-approximations — sound static analysis may identify false security risks, while a complete one may miss real ones. Dynamic taint analysis, identifying risks at runtime can be more precise.

In a manner akin to security and privacy taint analysis, we observe that issues of point-of-trust propagation in smart contracts (and indeed other systems which depend on data by external parties) follow a similar pattern and can be addressed using similar tools. We envisage trust type checking to ensure that trust does not propagate in an unexpected manner as a primary tool for smart contracts enforcing business process flows dependent on oracle and user data. Furthermore, we believe that the notion of trust is core in smart contracts and by implementing trust at the programming language level, and allowing developers to use trust information as part of their logic can be of great benefit. For this reason, we ensure that our trust/taint propagation semantics extend the semantics of Solidity, and allow for dynamic execution.

Static taint analysis has been explored in the context of Solidity smart contracts, e.g. [17, 10], however to the best of our knowledge dynamic taint analysis

has not, possibly due to the associated overheads. In fact, both deploying and executing functions of the smart contract costs *gas*, paid for in *ether*, the currency of the Ethereum blockchain. This cost can be an effective remedy against denial-of-service attacks, and also ensures termination, but discourages the use of dynamic analysis techniques on the blockchain. Static analysis however has been used to attenuate the gas overheads of runtime verification (e.g., see [5]).

In this paper, we present an extension of Solidity with a notion of tainting as a first-class concept. We present taint labels that carry data, and allow assertions in the language to query these taints and use the associated data. We give the semantics for a simplified version of Solidity with taints that propagates the taints. We give several static analyses that we use to prune taint instrumentation, and leave the remaining for runtime. As a formal basis for these analyses and future ones, we give an abstract sound static semantics for the language.

Related Work. Static taint analysis has been applied in the context of Solidity before. Slither [10] can classify whether a smart contract variable is dependent on a user-controlled variable (e.g., a function parameter), or whether a function can be entered from illegitimate entry points. [17] uses taint tracking on control-flow graphs to identify re-entrancy. Our work instead considers using dynamic analysis, and optimises it through static analysis. Such combined analysis have been applied in other contexts, such as web security (e.g, see [13, 16]), and Android applications (e.g., see [16, 14]).

Static analysis has been used before to prove parts of properties safe and leave the rest of the property for runtime [4, 2, 8], and also for the pruning of instrumentation [3], mainly in the context of Java verification. This work has also been applied in the context of Solidity verification [5, 6]. See [11] for a more general exposition of optimisations for monitors.

See [15] for a more general survey of formal verification techniques applied to smart contracts.

Summary. In Sec. 2 we present an extension of the Solidity language with taints and a semantics for it, while in Sec. 3 we present tools to statically analyse programs in this language. We present a case study to validate the example static analyses we give in Sec. 4. We discuss this approach in Sec. 5, while we conclude with future work in Sec. 6.

2 Solidity with Dynamic Tainting

We present an extension of Solidity with taints at the language level, including constructs for declaration of data-carrying taint labels, statements that taint variables or memory locations, and extend Boolean expressions to query taints.

2.1 Simplified Solidity with Taints

The grammar of Solidity extended with taints is shown in Fig. 1, with our additions and modification in **boldface**.

TaintLabel := **Label**
TaintExpr := **TaintLabel** | **TaintLabel [type Label?]** | **TaintExpr or TaintExpr**
TaintValue := **TaintLabel** | **TaintLabel [VarLabel** | **Value]**
TaintDeclr := ***newtaint*** **Label** = **TaintExpr**
TaintQuery := ***taint-of*** **VarLabel**

BoolTaintsExpr := **BoolExpr** | **VarLabel tainted-by [TaintValue** | **TaintQuery]** | **!BoolTaintsExpr** | **BoolTaintsExpr && BoolTaintsExpr** | **BoolTaintsExpr || BoolTaintsExpr**
CallExpr := Call Label [Values] | GuardedCall Label [Values]
Expr := Label | CallExpr | ValueExpr
Declr := type Label
TaintBy := ***taint*** **VarLabel** ***by*** **[TaintValue]**
Assign := **Label** $\overset{\checkmark}{=}$ **Expr** | **Label** $\overset{\times}{=}$ **Expr**
Assert := *assert* **BoolTaintsExpr**
Stmt := Declr | Assign | Return Expr | *revert* | **TaintBy** | Assert | *If* **BoolTaintsExpr** *then* [Stmt] *else* [Stmt] | *While* BoolExpr *then* [Stmt]
visibility := *public* | *private* | *internal* | *external*
Func := *function* Label ([type Label]) (*returns* [type Label?]?) { [Stmt] }
Contract := *contract* Label { [(Declr | StructDeclr | **TaintDeclr**)] [Function] }

Fig. 1. Solidity with taints.

We leave the grammar underspecified for simplicity (e.g., we do not list *type*s, or all possible *ValueExpr*essions, like arithmetic combinations), so that we can focus on the novel taint constructs (see [9] for the full Solidity language). Smart contracts are deployed on the blockchain to certain addresses, and calls to their functions also are initiated from addresses, however here we abstract away from these, e.g. in *CallExpr* — note how a function being available from a certain address can simply be encoded as part of the function name/label; and also from messages (carrying some standard information about the sender and call), which can be encoded as parameters to the function.

We define taint template expressions (**TaintExpr**) to be either simple labels, labels with a sequence of possibly labelled data types, or disjunctions of such labels. We specify taint values (**TaintValue**) as being either simple labels (**TaintLabel**), or data-carrying labels (e.g., `BadActor (address location)` is a taint label template that can be instantiated into taints that carry information about the address of the bad actor). We introduce constructs to assign a taint expression a label (**TaintDeclr**), and a construct to allow for the taint of a variable to be a queried (**TaintQuery**).

We augment boolean expressions (**BoolTaintsExpr**) to be able to query the taints of variables, e.g. v `tainted-by BadActor` will hold true only if the value of v depends on some past interaction started by a specific bad actor. These can be used in `assert` statements (here we do not model gas consumption, thus we ignore `require` statements) and `if` statements.

Crucially, we add a construct (**TaintBy**) to allow variables to be assigned taint values, e.g., `taint` v `by BadActor msg.sender`[4]. Essentially, the user can use this construct to specify sources of taint, e.g., to taint some parameters at the start of a function definition. Propagation of these taints to any variables that in turn depends on tainted variables will be taken care of by the semantics.

[4] `msg.sender` in Solidity refers to the address (a unique identifier) of the function caller.

Moreover, instead of the assignment symbol $=$, we have two kinds of assignment symbols: (1) $x \overset{\checkmark}{=} expr$ denotes that x is assigned the value of the expression and also propagates taints of $expr$ to x; while (2) $x \overset{\times}{=} expr$ denotes that x is only assigned the value of $expr$. We do not intend this to be used by the user, but we use it to denote the instrumentation required for propagating tainting dynamically. From the point of view of the user they will use $=$, which will be interpreted as $\overset{\checkmark}{=}$. For our static analysis the aim is to turn as many $\overset{\checkmark}{=}$ statements into $\overset{\times}{=}$ while preserving the semantics. We will use $\overset{*}{=}$ to denote either $\overset{\checkmark}{=}$ or $\overset{\times}{=}$.

2.2 Semantics

We present an operational semantics for the grammar in Fig. 1, with some preliminaries first.

Preliminaries For brevity, we assume that every smart contract on the blockchain has unique names for their global variables, function parameters, local function variables, and function names.

The semantics given is an operational semantics, over configurations and transitions. Configurations are given over variable valuations, a function call stack, and the function code. The function call stack maintains a stack of the function calls in the current transaction, and a sequence of statements with the first statement being the next statement to execute. Instead of Solidity statements, we consider *tainted statements*, which will be required to keep track of taint of a certain execution, e.g., due to branching.

Definition 1 (Tainted Statements). *Given a statement st and a set of taints T, a* tainted statement*, written $st\#T$, denotes that the execution of st was tainted by T. We overload this to sequences of statements, such that $(st : sts)\#T \overset{\text{def}}{=} (st\#T : sts\#T)$ and $[]\#T \overset{\text{def}}{=} []$. We interpret $(st\#T)\#T'$ as $st\#(T \cup T')$.*

Definition 2 (Configurations). *A **configuration** is a triple $\langle \boldsymbol{V}, calls, \boldsymbol{F} \rangle$ where:*

1. *$\boldsymbol{V}$ is a valuation, a mapping from variable names to their values and taints (we write $\boldsymbol{V}[x \mapsto v]$ to update the value of variable x, and $\boldsymbol{V}[x_{taint} \mapsto t]$ to update the taint of x;*
2. calls *is a function call stack, an array modelling the current function call stack, with values consisting of a pair of: tainted sequence of statements and a variable (and variable taint) valuation; and*
3. *$\boldsymbol{F}$ is the code, a mapping from function names to sequences of statements (we leave this implicit since, for simplicity, we do not allow it to change).*

Essentially, a configuration models the state (including taint state) of a blockchain at a given point in time. When the call stack is empty, the configuration is that of the blockchain between transactions, and when it is not empty the blockchain is in the process of a transaction.

The semantics will propagate a taint throughout a function's code, which may depend on the taint of a expression, which we define as the union of the taints of the variables mentioned in that expression.

Definition 3. *The* taint *of a value expression expr in the context of a valuation* $\boldsymbol{V}$*, denoted by* $taint(expr, \boldsymbol{V})$*, is the union of the set of taints associated with every variable mentioned in the expression. When the valuation is clear from the context we leave it implicit.*

We also require a notion of evaluation of expressions in the context of a given valuation of variables. We define an operator to represent this.

Definition 4. *Given a valuation* $\boldsymbol{V}$ *and Solidity expression expr, we write* $expr \Downarrow \boldsymbol{V}$ *to denote the value of the expression with respect to the valuation.*

We can then give the operational semantics. The notation we use for the operational semantics includes naming of certain structures for more compact (widthwise) rules, e.g., writing $lcls' := lcls[x \mapsto expr \Downarrow lcls]$ means $lcls'$ should be interpreted as $lcls[x \mapsto expr \Downarrow lcls]$ in the rest of the rule.

Definition 5 (Operational Semantics). *The* operational semantics of Solidity with taints *is given over configurations and transitions labelled by calls and tainted return values, or by* $\times$ *(denoting an unsuccessful call). The transition relation* $\rightarrow$ *is given by the rules in Fig. 2.*

We use $\Rightarrow$ *for the transitive closure of* $\rightarrow$*. We overload* $\Rightarrow$ *so that we write* $V \xrightarrow[res]{call(f,params)} V'$ *for* $(V, []) \xrightarrow{call(f,params)} (V, x) \Rightarrow (V, x') \xrightarrow{res} (V', [])$ *(with no other labelled transitions in between).*

Note that we do not define a rule for `assert`, instead we treat a statement `assert(e)` as `if(!e) revert(); else{rest of code}`.

We briefly describe the semantics. Labels indicate the start of an offchain call (OffchainCall) or the termination of such a call, either without exceptions (ReturnOffchain) or with a cancellation and revert of the transaction (RevertOffchain). Given a tainting expression, we taint the variable in the valuation (UpdateTaint), while if the initiator of the call, `msg.sender`, is tainted then all the assignments in the remaining statements are also tainted (UpdateTaintSender). Given an if statement, we evaluate the condition on the current valuation, and continue in the appropriate branch, while tagging each branch with the taints of the condition (IfThenElseLeft, IfThenElseRight).

Given an assignment, we first consider when the right-hand side expression is a value expression and update the value of the variable (NonCallAssignment), and in the case the instrumented assignment, is used the taint of the variable is also updated (NonCallAssignmentInstrumented). When there is a call, we simply place the code of the called function on the stack (Call), note how our assumption that all variables, parameters, and functions have unique names ensures the valuation is updated appropriately. The output of a function is then

used if the call ends successfully (CallReturn) and the taint possibly updated (CallReturnInstrumented), or otherwise a `revert` is propagated upwards (CallRevert). This logic is modified slightly for the case of a guarded call (GCall, GCallReturn, GCallReturnInstrumented, and GCallRevert), where `revert`s no longer propagate upwards.

2.3 Implementation

Implementing this semantics as is requires augmenting the semantics of Solidity. Instead, here, we describe how it can be encoded in the full Solidity language.

Taints values. Taint values can be encoded with each taint label as a value in an `enum`, and a wrapping `struct` as a template for values. For example, the declaration `BadActorTaint = BadActorUnknown | BadActor (address loc)` can be represented with: `enum BadActorTaintLabels = {BadActorUnknown, BadActor}` and `struct BadActorTaint = {BadActorTaintLabels label; address loc;}`.

Variable Taints. The taint of each variable can be kept track of in a corresponding taint array variable. The tainting of locations in a mapping or array can also be kept track of in variables of the same structure. A taint expression `taint` x `by` t, can then encoded by simply pushing taint t onto x's taint array, e.g., `xTaint.push(t)`. We can have repetition in this case, i.e. `xTaint` is not a set, but this does not change the semantics.

Propagating Taints. Propagating taints through instrumented assignments can be done in two-steps. For direct assignments to a value expression, one can simply append a statement immediately after the assignment that sets the taint of the assigned variable to the union of the taint variables of the variables used in the assignment expression. Assignment to the value of calls however presents an issue. If the function called is under our control, we can simply edit it to take parameter taints as inputs and to output also the taints of the return values. Otherwise[5], the taint semantics cannot be replicated. One approach could be to assume that the output could be tainted by any taint, and thus have a sound but incomplete dynamic taint analysis. For our purposes, we only consider when called functions are under our control and then the analysis is sound.

This approach to the implementation can however make the smart contract very costly (note how checking a taint query requires iterating over an array which does not have a bounded size). We tackle this through static analysis.

3 Static Analysis

When developing smart contracts one generally aims to reduce the amount of code and the amount of computation performed. This is due to the notion of *gas*,

[5] If we do not know the code behind a function call we cannot determine the possible taint of return values.

$$\frac{}{(\mathbf{V}, []) \xrightarrow{call(f, params)} (\mathbf{V}, [(\mathbf{F}(f), params \cup \mathbf{V})])} \textsc{OffchainCall}$$

$$\frac{\mathbf{V}' = \mathbf{V}[v \mapsto lcls(v)]}{(\mathbf{V}, ((return\ expr : sts)\#T, lcls)) \xrightarrow{(expr \Downarrow lcls, T \cup taint(expr))} (\mathbf{V}', [])} \textsc{ReturnOffchain}$$

$$\frac{}{(\mathbf{V}, ((revert : sts)\#T, lcls)) \xrightarrow{\times} (\mathbf{V}, [])} \textsc{RevertOffchain}$$

$$\frac{lcls' := lcls[x_{taint} \mapsto taintExpr \Downarrow lcls]}{(\mathbf{V}, ((taint\ x\ by\ taintExpr) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, (sts\#T, lcls') : rest)} \textsc{UpdateTaint}$$

$$\frac{T' := taintExpr \Downarrow lcls}{(\mathbf{V}, (((taint\ msg.sender\ by\ taintExpr) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, (sts\#T', lcls) : rest)} \textsc{UpdateTaintSender}$$

$$\frac{c \Downarrow lcls}{(\mathbf{V}, (((if\ c\ then\ sts_1\ else\ sts_2) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, ((sts_1\#(taint(c)) +\!\!+ sts)\#T, lcls) : rest)} \textsc{IfThenElseLeft}$$

$$\frac{\neg c \Downarrow lcls}{(\mathbf{V}, ((if\ c\ then\ sts_1\ else\ sts_2) : sts\#T, lcls) : rest) \rightarrow (\mathbf{V}, ((sts_2\#(taint(c)) +\!\!+ sts)\#T, lcls) : rest)} \textsc{IfThenElseRight}$$

$$\frac{c \Downarrow lcls}{(\mathbf{V}, (((while\ c\ \{sts'\}) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, (sts'\#(taint(c)) +\!\!+ (while\ c\ \{sts'\} : sts))\#T, lcls) : rest)} \textsc{WhileEntry}$$

$$\frac{\neg c \Downarrow lcls}{(\mathbf{V}, (((while\ c\ \{sts'\}) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, (sts\#T, lcls) : rest)} \textsc{WhileExit}$$

$$\frac{lcls' := lcls[x \mapsto expr \Downarrow lcls]}{(\mathbf{V}, (((x \overset{\times}{=} expr) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, (sts\#T, lcls') : rest)} \textsc{NonCallAssignment}$$

$$\frac{lcls' := lcls[x \mapsto expr \Downarrow lcls][x_{taint} \mapsto T \cup taint(expr)]}{(\mathbf{V}, (((x \overset{\checkmark}{=} expr) : sts)\#T, lcls) : rest) \rightarrow (\mathbf{V}, (sts\#T, lcls') : rest)} \textsc{NonCallAssignmentInstrumented}$$

$$\frac{calls := (((x \overset{*}{=} Call(f', params')) : sts)\#T, lcls : rest)}{(\mathbf{V}, calls) \rightarrow (\mathbf{V}, ((\mathbf{F}(f')\#T, params' \cup params'_{taints} \cup lcls) : calls))} \textsc{Call}$$

$$\frac{\begin{array}{c} calls := (((x \overset{\times}{=} Call(f', params')) : sts)\#T, lcls') : rest \\ lcls'' := lcls[v \notin dom(\mathbf{V}) \mapsto lcls'(v)][x \mapsto expr \Downarrow lcls'] \end{array}}{(\mathbf{V}, ((return\ expr : sts')\#T', lcls) : calls) \rightarrow (\mathbf{V}, (sts\#T, lcls'') : rest)} \textsc{CallReturn}$$

$$\frac{\begin{array}{c} calls := (((x \overset{\checkmark}{=} Call(f', params')) : sts)\#T, lcls') : rest \\ lcls'' := lcls[v \notin dom(\mathbf{V}) \mapsto lcls'(v)][x \mapsto expr \Downarrow lcls'][x_{taint} \mapsto T \cup T' \cup taint(expr)] \end{array}}{(\mathbf{V}, ((return\ expr : sts')\#T', lcls) : calls) \rightarrow (\mathbf{V}, (sts\#T, lcls'') : rest)} \textsc{CallReturnInstrumented}$$

$$\frac{calls := (((x \overset{*}{=} Call(f', params')) : sts)\#T, lcls' : rest)}{(\mathbf{V}, ((revert : sts)\#T', lcls) : calls) \rightarrow (\mathbf{V}, (sts\#T, lcls') : rest')} \textsc{CallRevert}$$

$$\frac{calls := (((success, x) \overset{*}{=} (guardedcall(f', params')) : sts)\#T, lcls : rest)}{(\mathbf{V}, calls) \rightarrow (\mathbf{V}, ((\mathbf{F}(f')\#T, params' \cup lcls) : calls))} \textsc{GCall}$$

$$\frac{\begin{array}{c} calls := (((success, x) \overset{\times}{=} (guardedcall(f', params')) : sts)\#T, lcls') : rest \\ lcls'' := lcls[v \in dom(\mathbf{V}) \mapsto lcls'(v)][x \mapsto expr \Downarrow lcls'][success \mapsto true] \end{array}}{(\mathbf{V}, ((return\ expr : sts')\#T', lcls) : calls) \rightarrow (\mathbf{V}, (sts\#T, lcls'') : rest)} \textsc{GCallReturn}$$

$$\frac{\begin{array}{c} calls := (((success, x) \overset{\checkmark}{=} (guardedcall(f', params')) : sts)\#T, lcls') : rest \\ lcls'' := lcls[v \in dom(\mathbf{V}) \mapsto lcls'(v)][x \mapsto expr \Downarrow lcls'][x_{taint} \mapsto T \cup T' \cup taint(expr)][success \mapsto true] \end{array}}{(\mathbf{V}, ((return\ expr : sts')\#T', lcls) : calls) \rightarrow (\mathbf{V}, (sts\#T, lcls'') : rest)} \textsc{GCallReturnInstrumented}$$

$$\frac{\begin{array}{c} calls := (((success, x) \overset{*}{=} (guardedcall(f', params')) : sts)\#T, lcls') : rest \\ lcls'' := lcls'[success \mapsto false] \end{array}}{(\mathbf{V}, (((revert) : sts)\#T', lcls) : calls) \rightarrow (\mathbf{V}, (sts\#T, lcls'') : rest)} \textsc{GCallRevert}$$

Fig. 2. Semantics of grammar in Fig. 1.

wherein both placing code on the blockchain and executing it has costs. Our described implementation, however, requires adding substantial instrumentation: (1) a taint variable for every variable; and (2) assignment to these taint variables after every assignment to associated variables. These can add significant overheads, as we see later in Sec. 4. Yet, not all this instrumentation is required and tainting is only relevant to the smart contract's execution when it affects the flow or output of the smart contract, otherwise it has no impact.

In this section, we give a sound abstract semantics to the language which we use as the basis for static analysis that is able to modify instrumentation safely (e.g., transform $\overset{\checkmark}{=}$ into $\overset{\times}{=}$), and that can be used to determine the possible value of conditions on taint at locations of a smart contract.

3.1 Abstract Semantics

Here we define a sound method of propagating taints in a Solidity smart contract, while abstracting away the values of variables.

In the static context we do not have taint values when the taint is data-carrying, instead we abstract them by their corresponding taint expression, e.g., `BadActor(msg.sender)` is abstracted by the expression `BadActor address`. In this section, we then use these taint expressions as our taints.

Definition 6 (Abstract Taints). *The* abstraction of a taint tag t*, denoted* $abs(t)$*, is* t *itself in the case of a non-data-carrying labels, and the corresponding taint expression with values replaced by their types for data-carrying labels. We overload abs to also range over sets of taints, i.e.,* $abs(T) \overset{\text{def}}{=} \{abs(t) \mid t \in T\}$.

We will also require a notion of projecting concrete valuations and return values onto their original variable value parts and the taint parts.

Definition 7 (Valuation and Return Value Projection). $\boldsymbol{V}|_{vars}$ projects $\boldsymbol{V}$ *onto its variable domain, ignoring tainting.* $ret|_{vars}$ *is similar, while* $\times$ *remains* $\times$*. Similarly,* $|_{taints}$ *projects* $\boldsymbol{V}$ *and* ret *onto taint variables.*

A remaining issue is branching in the code as caused by an if-then-else, where the taint at runtime depends on which branch is taken. Statically we have to consider both branches, since we want to handle every possible case. Since we will be dealing with each function on its own, and Solidity smart contracts have a tendency to be small (due to gas costs), here we will deal with this simply by non-deterministically branching. In other contexts this may not be ideal, since this may incur a degree of repetition which may worsen the state space explosion.

We re-use the # and *taint* operators here, appropriately re-interpreted for this abstract context (i.e. # instruments with abstract taints, and *taint* returns the abstract taints of an expression).

The semantics we give is over abstract configurations, which only maintain information about the next statement to execute and an abstract taint function.

Definition 8 (Abstract Configurations). *An* ***abstract configuration*** *is a pair* $\langle calls, tnts, \boldsymbol{F} \rangle$ *with:*

1. calls *is an abstract call stack, with elements as abstractly tainted statements;*
2. tnts *is an abstract taint valuation; and*
3. $\boldsymbol{F}$*, being the code, a mapping from function names to the function's list of statements (left implicit).*

We can then give our abstract operational semantics.

Definition 9 (Abstract Operational Semantics). *The* abstract operational semantics *of Solidity with taints is given over abstract configurations. The transition relation* $\rightarrow$*, is given by the rules in Fig. 3, with* $\Rrightarrow$ *as its transitive closure.*

Every rule given is a direct counterpart of the similarly named rules in Fig. 2, with some rules combined into one here.

We can prove that this abstract semantics soundly abstracts the concrete semantics of the language, i.e., that when a call produces a return value with a certain concrete taint in the concrete semantics, then there is a path in the abstract semantics that produces an abstract version of the concrete taint.

Theorem 1. $(\boldsymbol{V}, []) \xRightarrow[(expr \Downarrow V', T)]{call(f, params)} \boldsymbol{V'}$ *implies* $\exists sts, T', tnts' \cdot (\boldsymbol{F}(f),\ \boldsymbol{V}|_{taints}) \Rrightarrow ((return\ expr : sts)\#T', tnts') \wedge abs(T) = T' \cup taint(expr, tnts')$.

3.2 Analysis and Optimisation

The abstract semantics we gave can be the formal basis of different static analyses. Here we characterise when a static analysis reduces instrumentation in a correct manner. However, instead of working with the code, for static analysis it is often more useful to make the control-flow between statements explicit. Standard techniques can be used to transform Solidity code into a graph and back (e.g., as supported by existing tools [6, 5]).

Definition 10 (Function Control-flow Graph). *The* control-flow graph of a function F *is a tuple* $C_F = \langle S, label, s_0, Ret, Rev, \rightarrow \rangle$*, with* S *being a set of states,* $label : S \rightarrow Stmt$ *associating each state with a statement,* s_0 *being the initial state,* Ret *being the set of states associated with return statements, and* Rev *being the set of states associated with revert statements.* $\rightarrow$*:* $S \times S$ *is a transition relation denoting the control-flow between the statements.*

We can then augment the control-flow graph by considering its abstract execution with the abstract semantics. The states in the graph then become pairs of statements and abstract taint functions.

Moreover, we consider a special abstract taint expression $*$ that denotes variables could be tainted by any taint set; we will be using this to be able to reason about each function *intraprocedurally*, by starting with an abstract taint function that assigns $*$ to every variable: $initTnt$, s.t. $initTnt(v) = *$.

$$\frac{}{(((taint\ x\ \ by\ T) : sts)\#T', tnts) \rightarrow (sts\#T', tnts[x \mapsto T \cup T'])}\ \textsc{AUpdateTaint}$$

$$\frac{}{(((taint\ msg.sender\ \ by\ T) : sts)\#T', tnts) \rightarrow (sts\#(T \cup T'), tnts)}\ \textsc{AUpdateTaintSender}$$

$$\frac{}{(((x \stackrel{\times}{=} expr) : sts)\#T, tnts) \rightarrow (sts\#T, tnts)}\ \textsc{ANonCallAssignment}$$

$$\frac{tnts' = tnts[x_{taint} \mapsto T \cup taints(expr)]}{(((x \stackrel{\checkmark}{=} expr) : sts)\#T, tnts) \rightarrow (sts\#T, tnts')}\ \textsc{ANonCallAssignmentInstrumented}$$

$$\frac{}{\begin{array}{l}(((if\ c\ then\ sts_1\ \ else\ sts_2) : sts)\#T, tnts) \rightarrow ((sts_1\#taint(c) +\!\!+ sts)\#T, tnts) \\ (((if\ c\ then\ sts_1\ \ else\ sts_2) : sts)\#T, tnts) \rightarrow ((sts_2\#taint(c) +\!\!+ sts)\#T, tnts)\end{array}}\ \textsc{AIfThenElse}$$

$$\frac{}{\begin{array}{c}(((while\ c\ \{sts'\}) : sts)\#T, tnts) \rightarrow (((sts'\#taint(c) +\!\!+ ((while\ c\ \{sts'\}) : sts))\#T, tnts) \\ (((while\ c\ \{sts'\}) : sts)\#T, tnts) \rightarrow (sts\#T, tnts)\end{array}}\ \textsc{AWhile}$$

$$\frac{\begin{array}{c}call := (((x \stackrel{\times}{=} Call(f', params')) : sts)\#T, tnts) \\ (\mathbf{F}(f')\#T, tnts) \rightarrow ((return\ expr : sts)\#T', tnts')\end{array}}{call \rightarrow (sts\#T, tnts')}\ \textsc{ACallReturn}$$

$$\frac{\begin{array}{c}call := (((x \stackrel{\checkmark}{=} Call(f', params')) : sts)\#T, tnts) \\ (\mathbf{F}(f')\#T, tnts) \rightarrow ((return\ expr : sts)\#T', tnts')\end{array}}{call \rightarrow (sts\#T, tnts'[x \mapsto T \cup T' \cup tnts(expr)])}\ \textsc{ACallReturnInstrumented}$$

$$\frac{\begin{array}{c}call := (((x \stackrel{*}{=} Call(f', params')) : sts)\#T, tnts) \\ (\mathbf{F}(f')\#T, tnts) \rightarrow ((revert : sts)\#T', tnts')\end{array}}{call \rightarrow (revert\#T, tnts')}\ \textsc{ACallRevert}$$

$$\frac{\begin{array}{c}call := ((((success, x) \stackrel{\times}{=} GuardedCall(f', params')) : sts)\#T, tnts) \\ (\mathbf{F}(f')\#T, tnts) \rightarrow ((return\ expr : sts)\#T', tnts')\end{array}}{call \rightarrow (sts\#T, tnts')}\ \textsc{AGCallReturn}$$

$$\frac{\begin{array}{c}call := ((((success, x) \stackrel{\checkmark}{=} GuardedCall(f', params')) : sts)\#T, tnts) \\ (\mathbf{F}(f')\#T, tnts) \rightarrow ((return\ expr : sts)\#T', tnts')\end{array}}{call \rightarrow (sts\#T, tnts'[x \mapsto T \cup T' \cup tnts'(expr)])}\ \textsc{AGCallReturnInstrumented}$$

$$\frac{\begin{array}{c}call := (((x \stackrel{*}{=} Call(f', params')) : sts)\#T, tnts) \\ (\mathbf{F}(f')\#T, tnts) \rightarrow ((revert : sts)\#T', tnts')\end{array}}{call \rightarrow (sts\#T, tnts')}\ \textsc{AGCallRevert}$$

Fig. 3. Abstract static semantics of grammar in Fig. 1.

Definition 11 (Tainted Function Control-flow Graph). *The* tainted control-flow graph of a function F *is a tuple* $t(C_F) = \langle S, tlabel, s_0, Ret, Rev, \rightarrow \rangle$*, defined as before, but with* $tlabel : S \rightarrow Stmt \times \mathbb{V}_{taints}$ *associating each state with a statement and an abstract taint function.*

The construction proceeds by associating the initial state with the most abstract taint function, $tlabel(s_0) = (st_0, initTnt)$*, and when for states* s *and* s'*,* $s \rightarrow s'$ *in* C_F*, then if* $tlabel(s) = (label(s), tnt)$ *and* $(label(s) : [label(s')], tnt) \rightarrow (label(s'), tnt')$ *(in the abstract semantics), we set* $tlabel(s') = (label(s'), tnt')$*.*

Note how we have a finite amount of abstract taints and statements, and thus applying the abstract semantics will terminate, if there is no recursive call. In the case of a recursive call we have several options, e.g., tainting with $*$, or running the call until a fixed point of taints is reached.

Our static analysis will involve transformation of the instrumentation of a function while retaining the same semantics, which we characterise below.

Definition 12 (Instrumentation Reduction). *Given functions* F *and* F'*,* F' *is said to be an* instrumentation reduction *of a function* F*, in the context of a set of functions* $\boldsymbol{F}$*, written* $F' \leq_t F$ *iff (1)* F *and* F' *only differ on the use of* $\overset{\checkmark}{=}$ *or* $\overset{\times}{=}$*, or on the presence of taint by expressions; (2) replacing a call to* F *with one to* F' *does not change the values of variables, but may associate less (but not different) taints to variables, formally:*

For an arbitrary n*, consider any arbitrary sequence of* n *function calls (to functions from* $\boldsymbol{F}$*),* c_i*, any initial valuation* $\boldsymbol{V}_0$*, and the corresponding* n *valuations* $\boldsymbol{V}_{i+1}$ *and return values* ret_{i+1}*, such that* $\boldsymbol{V}_i \xRightarrow[ret_{i+1}]{c_i} \boldsymbol{V}_{i+1}$*. If, for some index* j*,* c_j *is a call to* F*, then replacing* c_j *with* c'_j*, a call to* F' *but with the same parameters and message, induces* $n - j$ *new valuations and return values* $\boldsymbol{V}'_{j+1}, ..., \boldsymbol{V}'_{n+1}$ *and* $ret'_{j+1}, ..., ret'_{n+1}$ *such that* $\boldsymbol{V}_j \xRightarrow[ret'_{j+1}]{c'_j} \boldsymbol{V}'_{j+1}$ *(and so on), then for all indices* k *bigger than* j *the corresponding valuations and return values with taints projected out, are equivalent:* $\boldsymbol{V}_k|_{vars} = \boldsymbol{V}'_k|_{vars}$ *and* $ret_k|_{vars} = ret'_k|_{vars}$ *, while the taint projected parts in the reduced version are contained in the other:* $\forall v \cdot \boldsymbol{V}'_k|_{taints}(v) \subseteq \boldsymbol{V}_k|_{taints}(v)$ *and* $ret'_k|_{taints} \subseteq ret_k|_{taints}$*.*

We then describe informally several instrumentation reducing analyses that can be performed on the set of tainted control-flow graphs of a smart contract.
Removing Irrelevant Instrumentation Given a function $F \in \mathbf{F}$, we can identify statements in that function whose evaluation depends on the taint of some variable, generally either conditional or return statements. For each such statement, we can do a transitive backwards analysis to determine the set of taints and the set of variables that are relevant.

Then, collecting all this information from all the functions in $\mathbf{F}$, we can identify the *taint instrumentation nodes* that set the taint of a variable such that the variable and its taint may be relevant to some conditional statement in the smart contract. Irrelevant taint instrumentation nodes can be removed.

For example, where T and T' are distinct concrete taint labels:

```
taint v by T'; x ≙✓ v;
assert(x tainted-by T);
```

$\leq_t$

```
x ≙× v; assert(x tainted-by
T);
```

On the left-hand side, x is relevant to the conditional on the third line, but it is only relevant to it when x is tainted by T. Thus, barring any other need for knowing about the taint of v or x with T', the right-hand is equivalent to the above modulo the `assert` statement.

Moreover, consider that a variable is tainted twice in a function with such a label, and between these two locations there is no point where the first taint of the variable is used. Then we can just discard the first instrumentation, and keep the last one, since the first one is unused and later overwritten.

For example (assume the only conditional statement is the visible `assert`):

```
uint v, x, y;
...
x ≙✓ v;
...
x ≙✓ 7*y;
assert(x tainted-by T);
```

$\leq_t$

```
uint v, x, y;
...
x ≙× v;
...
x ≙✓ 7*y;
assert(x tainted-by T);
```

Here, if y does not also depend previously on x, we can simply turn off the first tainting of x, since it will later be overwritten.

Push Forward Instrumentation Instrumentation points can set the taint of a variable v to that of another variable v'. We observe that sometimes the taint of variable v' is only relevant because it is relevant for v. However, if the taint instrumentation in question is in the same function we simply replace the reference to the taint of v' in the instrumentation of v by the concrete taint instrumentation of v'. Then the tainting of v' can be removed as in the previous optimisation. This concretisation can be performed easily, without any restrictions, for non-data-carrying labels. However, in the case of data-carrying labels we need to also make sure that the label does not contain references to variables that are modified in the flow between v' and v.

For example:

```
taint par by T;
uint v, v';
v' ≙✓ /* an operation on par */;
v ≙✓ /* an operation on v' */;
assert(v tainted-by T);
```

$\leq_t$

```
uint v, v';
v' ≙× /* an operation on par */;
v ≙× /* an operation on v' */;
taint v by T;
assert(v tainted-by T);
```

If the left-hand side is the whole body of a function, or we know that the parameter *par* and v' are not relevant to any other conditional statement, then simply removing their tainting instrumentation, and simply tainting v will be an appropriate reduction.

Table 1. Results, with costs in gas and increases in percentage w.r.t. to original costs.

Costs	**Original**	**Tainted**		**Optimised**	
	gas	*gas*	*%*	*gas*	*%*
Deployment	1276798	2213484	+73%	1694698	+32%
recordGrapeProductionFarmer Call	131783	312792	+137%	176743	+34%
recordGrapeProductionLab Call	152570	299379	+96%	242388	+58%
updateGrapeProductionLab Call	106448	121014	+13%	115372	+8%
Average Business Flow	225814	445715	+97%	300747	+33%

These two optimisations can be performed on the smart contract until a fixpoint is reached. We next consider the savings these give with a case study.

4 Case Study

We consider a smart contract which can be used to record and authenticate the provenance, quality, and use of grapes for the production of wine. Fig. 4 illustrates a selection of the functions of this smart contract, along with taint annotations in our language, in **boldface**. Note that for simplicity in this case study a variable can only have one taint.

This contract allows a farmer to record a grape production on the smart contract, which is given a certain unique identifier (`recordGrapeProductionFarmer`). We also allow accredited labs to either register a grape production themselves, or update the farmer record, which we do not show here since they are similar to the `recordGrapeProductionFarmer` function. Sale of grapes to another person is also recorded on the blockchain (`recordSale`). The owner of a certain grape production can then record the bottling of wine produced from grapes they own (`RecordBottling`), while official certification providers can give a certain certification to the grapes, depending on the location they are produced in.

In this smart contract, we are interested in specifying that the right kind of accredited lab was involved in determining the recording of a grape production involved in making a wine, depending on whether the wine involves multiple grape sources or just one (see the asserts over taints in `recordDOK`). It bears to note that the benefit of taints here is that propagation of taints is done automatically, while a manual ad hoc approach is open for errors.

In Fig. 5 we report the `recordGrapeProductionFarmer` function with the encoding in Solidity described in Sec. 2.3. After the optimisations described in Sec. 3.2, the result is shown in Fig. 6, a significant reduction.

We evaluated this smart contract to identify the gas costs when there are no taints, to when the taint instrumentation is performed, and after it is optimised. The results are shown in Table. 1. We report the results for each individual function given expected input, and for the average gas cost given a set of randomly generated expected (i.e., non-reverting) flows. One can see optimisation through the presented static analyses reduces costs significantly, up to around two thirds in the case of an average flow, validating the viability of the approach.

```
newtaint Taint = Farmer [address,uint] | Lab [address sender, uint]

function recordGrapeProductionFarmer(uint varietyOfGrapes, uint date, Location productionLoc,
                          GrapeQuality calldata qualityParameters) public returns(uint){
    taint msg.sender by Farmer(msg.sender, block.timestamp)
    require(!accredFullAnalysisLabs[msg.sender] && !accredSimpleAnalysisLabs[msg.sender]);
    grapeIdCounter++;

    GrapeProduction memory grapeProduction =
        GrapeProduction(varietyOfGrapes,
                        msg.sender,
                        date,
                        productionLoc,
                        address(0),
                        qualityParameters,
                        0);

    grapeProd[grapeIdCounter] = grapeProduction;
    grapeToOwner[grapeIdCounter] = grapeProd[grapeIdCounter].farmer;
    return grapeIdCounter;
}

function recordSale(uint grapeid, address newOwner) public
  only(grapeToOwner[grapeid]){
    grapeToOwner[grapeid] = newOwner;
}

function recordBottling(uint[] calldata grapeids) public{
    wineIdCounter++;
    for(uint i = 0; i < grapeids.length; i++){
        assert(grapeToOwner[grapeids[i]] == msg.sender);
        wineToGrape[wineIdCounter].push(grapeids[i]);
    }
}

function recordDOK(uint wineID, DOKType dok, bool mixedWine) public{
    require(certificationProviders[msg.sender]);

    uint[] memory grapeSources = wineToGrape[wineID];

    if(mixedWine){
        assert(grapeSources.length > 1);
        for(uint i = 0; i < grapeSources.length; i++){
        assert(accredFullAnalysisLabs[taint-of(grapeProdTaints[grapeSources[i]])).sender]);
        }
    } else {
        for(uint i = 0; i < grapeSources.length; i++) {
        assert(accredFullAnalysisLabs[taint-of(grapeProdTaints[grapeSources[i]])).sender] ||
             accredSimpleAnalysisLabs[taint-of(grapeProdTaints[grapeSources[i]])).sender]);
        }
    }

    wineBatchCertified[wineID] = dok;
}
```

Fig. 4. Extract from case study smart contract (the.

```
function recordGrapeProductionFarmer(uint varietyOfGrapes, uint date, Location productionLoc,
                          GrapeQuality calldata qualityParameters) public returns(uint){
    require(!accredFullAnalysisLabs[msg.sender] && !accredSimpleAnalysisLabs[msg.sender]);
    senderTaint[msg.sender] = Taint(constructors.Farmer, msg.sender, block.timestamp);

    grapeIdCounter++;
    grapeIdCounterTaint = senderTaint[msg.sender];

    GrapeProduction memory grapeProduction =
        GrapeProduction(varietyOfGrapes,
                        msg.sender,
                        date,
                        productionLoc,
                        address(0),
                        qualityParameters,
                        0);
    Taint memory grapeProductionTaint = senderTaint[msg.sender];

    grapeProd[grapeIdCounter] = grapeProduction;
    grapeProdTaints[grapeIdCounter] = grapeProductionTaint;

    grapeToOwner[grapeIdCounter] = grapeProd[grapeIdCounter].farmer;
    grapeToOwnerTaints[grapeIdCounter] = grapeProdTaints[grapeIdCounter];
    return grapeIdCounter;
}
```

Fig. 5. Encoding of taint propagation in `recordGrapeProductionFarmer`.

```
function recordGrapeProductionFarmer(uint varietyOfGrapes, uint date, Location productionLoc,
                          GrapeQuality calldata qualityParameters) public returns(uint){
    require(!accredFullAnalysisLabs[msg.sender] && !accredSimpleAnalysisLabs[msg.sender]);

    grapeIdCounter++;

    GrapeProduction memory grapeProduction =
    GrapeProduction(varietyOfGrapes,
                        msg.sender,
                        date,
                        productionLoc,
                        address(0),
                        qualityParameters,
                        0);

    grapeProd[grapeIdCounter] = grapeProduction;
   grapeProdTaints[grapeIdCounter] = Taint(constructors.Farmer, msg.sender, block.timestamp);

    grapeToOwner[grapeIdCounter] = grapeProd[grapeIdCounter].farmer;
    return grapeIdCounter;
}
```

Fig. 6. `recordGrapeProductionFarmer` after optimisation of taint propagation.

5 Discussion

The taints we have added here are different from the usual taints considered in taint analysis. Usually, something is considered as tainted or not tainted, while here tainting can be with different labels, even sets of labels. This is a more powerful concept, since it allows to talk about all the contributing actors to a value, rather that simply talk about whether something is a security risk or not (without identifying what caused that security risk). That we integrate queries about these taints into a language, allowing branching in the program due to taints, is also novel to the best of our knowledge, since standard taint analysis simply is concerned with preventing certain data from reaching certain sinks.

These queries allow the developer to make decisions based on whether they trust the source of some data or not. For example in the case study the developer requires that information comes from a certain kind of lab when the wine is of a certain kind. This can certainly be implemented in an ad hoc manner without taints, but we believe that this kind of reasoning about trust at the top-level can be very useful due to the immutability of smart contracts and their public accessibility. A high-level approach gives certain guarantees that ad hoc implementations do not give, while static analysis tackles issues of gas.

Moreover, allowing taint queries at the level of if-then-else constructs opens up the possibility to modify branching at runtime depending on the trust level one has towards sources of taints. This can be used not just to prevent untrustworthy data to have an effect on the smart contract, but also to keep track of bad flows and perform actions that sanitise such data or to sanction their source.

6 Conclusions

A smart contract on a blockchain is open for interaction, with input coming from different, possibly untrustworthy sources. Keeping track of the sources of some data can be useful, for example when an event of interest happens we can then query the source of the event and contributing smart contract state, and make decisions based on that. In this paper, we have incorporated dynamic taint analysis for Solidity smart contracts through an extension of the language with a formal semantics, while we have described how this can be implemented in Solidity. We have also introduced a way to perform static tainting, which we use to prune away some of the instrumentation judged inconsequential for dynamic tainting. Evaluation on a case study, validates the static analysis as potentially eliminating a significant amount of overhead.

Future Work. In our abstract semantics we abstract taints by their corresponding taint type expression. In the future we want to consider also adding some information in the abstract taints to be able to relate them together, for example adding information about the line of code where the taint is created. Moreover, we do not do any analysis of variable values at the static level, however we intend to use techniques and tooling from [5, 6] to enable some abstraction of these, which would allow more fine-grained static taint analysis.

References

1. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12478, pp. 9–24. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_2
2. Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G.: A specification language for static and runtime verification of data and control properties. In: Bjørner, N., de Boer, F. (eds.) FM 2015: Formal Methods. pp. 108–125. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_8
3. Azzopardi, S., Colombo, C., Pace, G.: Clarva: Model-based residual verification of java programs. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020 (2020). https://doi.org/10.5220/0008966603520359
4. Azzopardi, S., Colombo, C., Pace, G.J.: Control-flow residual analysis for symbolic automata. In: Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017. EPTCS, vol. 254, pp. 29–43 (2017). https://doi.org/10.4204/EPTCS.254.3
5. Azzopardi, S., Colombo, C., Pace, G.J.: Model-based static and runtime verification for ethereum smart contracts. In: Model-Driven Engineering and Software Development - 8th International Conference, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, Revised Selected Papers. Communications in Computer and Information Science, vol. 1361, pp. 323–348. Springer (2020). https://doi.org/10.1007/978-3-030-67445-8_14
6. Azzopardi, S., Colombo, C., Pace, G.J.: A technique for automata-based verification with residual reasoning. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020. pp. 237–248. SCITEPRESS (2020). https://doi.org/10.5220/0008981902370248
7. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: Contractlarva and open challenges beyond. In: Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 113–137. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_8
8. Chimento, J.M., Ahrendt, W., Pace, G.J., Schneider, G.: Starvoors : A tool for combined static and runtime verification of java. In: Runtime Verification. pp. 297–305. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_21
9. Ethereum: Solidity. Online Documentation `http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html` (2016)
10. Feist, J., Greico, G., Groce, A.: Slither: A static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. p. 8–15. WETSEB '19, IEEE Press (2019). https://doi.org/10.1109/WETSEB.2019.00008
11. Jakobs, M.C., Mantel, H.: A unifying framework for dynamic monitoring and a taxonomy of optimizations. In: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles. pp. 72–92. Springer International Publishing, Cham (2020)

12. Kim, J., Kim, T., Im, E.G.: Survey of dynamic taint analysis. In: 2014 4th IEEE International Conference on Network Infrastructure and Digital Content. pp. 269–272 (2014). https://doi.org/10.1109/ICNIDC.2014.7000307
13. Kurniawan, A., Abbas, B.S., Trisetyarso, A., Isa, S.M.: Static taint analysis traversal with object oriented component for web file injection vulnerability pattern detection. Procedia Computer Science **135**, 596–605 (2018). https://doi.org/https://doi.org/10.1016/j.procs.2018.08.227, the 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life
14. Mumtaz, H., El-Alfy, E.S.M.: Critical review of static taint analysis of android applications for detecting information leakages. In: 2017 8th International Conference on Information Technology (ICIT). pp. 446–454 (2017). https://doi.org/10.1109/ICITECH.2017.8080041
15. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7) (jul 2021). https://doi.org/10.1145/3464421
16. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: Effective taint analysis of web applications. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 87–97. PLDI '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1542476.1542486
17. Xue, Y., Ma, M., Lin, Y., Sui, Y., Ye, J., Peng, T.: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. p. 1029–1040. ASE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3324884.3416553, https://doi.org/10.1145/3324884.3416553