Machine Learning for Security Vulnerability Detection in *Ethereum*'s Solidity
[Work under review. Do not circulate without authors' permission.]

Shashank Srikant and **Erik Hemberg** and **Una-May O'Reilly**
CSAIL, MIT
shash@mit.edu, ehemberg@mit.edu, unamay@csail.mit.edu

A domain-specific language (DSL) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains. Example : html

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.

A distributed ledger is a database that is consensually shared and synchronized across multiple sites, institutions or geographies. It allows transactions to have public "witnesses," thereby making a cyberattack more difficult. The participant at each node of the network can access the recordings shared across that network and can own an identical copy of it.

Abstract With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

Domain specific languages such as Solidity for Ethereum have emerged in distributed ledgers, and bugs and vulnerabilities unique to it threaten the integrity of programs written in them. We propose a machine learning approach to designing vulnerability detectors. We introduce a highly interpretable representation of programs, which captures intricate data and control dependencies between various program variables. We evaluate our proposed approach and representations on 66,470 functions written in Solidity *smart contracts*. Predictive models built on our representations are able to match the performance of symbolic analysis based tools on four different classes of vulnerabilities. Features selected by these models suggest the models indeed learn the inherent structure of vulnerable Solidity programs. To the best of our knowledge, this is the first attempt at training machine learning models on programs using informative representations to classify general vulnerabilities.

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

1 Introduction

Detecting bugs and vulnerabilities in programs has been an area of research for well over three decades (Wong et al. 2016). Despite the rich literature that addresses this topic, vulnerabilities and bugs that escape recognition persist and remain a concern for developers and enterprises. Preemptive detection of vulnerabilities before program execution is a challenging problem because it is hard to statically characterize a program. A pathological combination of control paths and data transformations has to be anticipated. Although a vulnerability(such as a buffer overflow) is easy to describe, (in general or specific terms), identifying a causal path leading up to it is hard. Vulnerabilities generally arise as a consequence of complex interactions between different parts of a program and largely occur because of a mismatch between what a developer expects the program to do and what it actually does.

Recent research efforts have studied and modeled statistical properties of corpora of programs. This has resulted in a number of interesting applications relevant to the software engineering community, like automatically refactoring programs and smart renaming of variables, *etc*. (see Section 7). We believe that vulnerabilities too can be inferred from a corpora of code. Such a data-driven approach will circumvent the need for experts to hand-craft conditions to detect vulnerabilities, and will instead enable automatically inferring properties of programs which cause these issues.

In this work, we study programs written in Solidity, a domain specific language designed to model *Ethereum*, a popular decentralized, distributed ledger. While domain specific languages are extremely powerful tools to express ideas in niche application areas, they pose challenges of their own. As a consequence of their specialized design, they are susceptible to vulnerabilities and bugs very unique to their design. For instance, vulnerabilities seen in PDF-rendering languages are very different from those seen in shell scripts (Tzermias et al. 2011). Further, tools to detect and prevent such issues take longer to develop owing to the small communities. Specifically, Solidity deals with modeling high-stakes transactions. Recent bugs and vulnerabilities exploited in this language have resulted in multi-million dollar losses (Luu et al. 2016).

TO SEARCH

The inherent complexity of programming informs us that working with the right representation of programs is critical to modeling them. To successfully model the presence of vulnerabilities it is important to capture and quantify the complex interactions of variables. The very few related works that have modeled programs to detect vulnerabilities have approached representation in two broad ways - they either treat programs as a bag of tokens or they pass their representation challenge on to a deep network to learn features. We believe that these approaches represent extremes. Token and surface level information, as we show in this work, fails to model complex tasks, while expecting a deep learning model to infer complex variable interactions demands an inordinate magnitude of program samples, particularly so for nascent DSLs. We present in this work a way to extract semantically rich features which capture complex variable interactions. We show how an abstract syntax tree of a program expresses control and data dependency information which can be extracted to accurately classify the presence of vulnerabilities.

Contributions We make the following contributions:

- We introduce a highly interpretable representation of programs. These representations capture intricate data and control dependencies of variables appearing in a program.

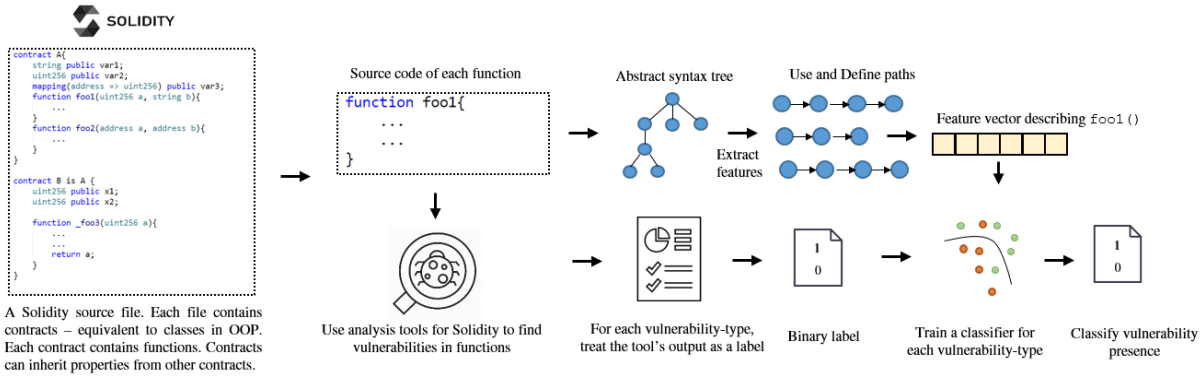

Figure 1: Our approach of designing a vulnerability classifier

LOOK AT LUU ET AL.2016

They capture the overall semantics and structure of programs and are hence independent of any specific class of vulnerabilities.
- Using these rich features, we build machine learning models of programs written in Solidity, and demonstrate these models' ability to predict four very diverse classes of vulnerabilities. Our engineered features enable us to successfully model in such low-data settings.
- We perform a qualitative analysis of the features selected by these models and show the high interpretability of our features allows for a causal understanding of the accuracy of our models.
- Through our qualitative analysis, we discover subtle issues in the state of the art vulnerability detection tools for Solidity.

2 Background

Solidity and *Ethereum*. *Ethereum* is a popular public, decentralized, distributed ledger. It allows for records to be maintained on it, allowing for transparency while maintaining immutability of its records. *Ethereum* specifically allows for 'programmable records', called *smart contracts* to be stored on its ledger. Smart contracts enable immutable programmable logic to be shared and executed by multiple parties. This has gained popularity in applications like deed signing, auctioning, *etc.*. Solidity is a nascent programming language which was designed for writing smart contracts. It follows an object-oriented paradigm, is statically typed, and compiles to bytecode which can be executed on *Ethereum*'s Virtual Machine (EVM). In Solidity's terminology, a *contract* corresponds to a class, where each class is described by inherent parameters like its *value* (currency it possesses), and its *owner* (specified by an address on the ledger), in addition to attributes defined by an application developer. Functions are written in a *contract* to conditionally perform actions depending on who the owner is, how much currency they posses, what time it is, *etc.*. Once a contract is *pushed* onto *Ethereum*, it is assigned a unique address and is immutable thereafter. Anyone can thereafter access and execute public functions and interfaces provided by the contract.

Vulnerabilities in Solidity programs. We analyze the following vulnerabilities seen in Solidity programs

- **Transaction order dependency (`TOD`).** This issue is specific to Solidity. Given Solidity's programming model, a caller calling a function can witness different program states depending on the order in which the function was called with respect to other functions in a contract. This allows a contract-writer to write malicious functions. See Section 3 and (Luu et al. 2016) for examples.
- **State change after execution (`StateChange`).** This issue is specific to Solidity as well. Solidity and the EVM does not support concurrent programming in that, when an external contract is called from within a contract, control-flow switches to the callee. As a consequence, if the callee code does not execute as expected, the function call gets stuck forever. If critical logic is coded *after* such a function call in the calling function, those lines of code may never get executed. This may lead to unintended consequences. For example, in Program 1, if `transfer` hangs because of a potential glitch in its code, then the function `close` would never be able to reset `isClose`, since it happens after the call to the erroneous `transfer`.
- **Integer Overflows, Underflows (`IntOv, IntUn`).** The Ethereum Virtual Machine (EVM) and Solidity's poor handling of integer overflows and underflows are well documented (Luu et al. 2016). In this class of bugs, a simple looking addition or subtraction operation does not produce intended results in cases when the result of the arithmetic operation is larger than the word-size assigned by EVM . As a consequence, an arithmetic operation being performed within a conditional statement, say, can result in unexpected behavior. This issue has been central to popular malicious hacks on the Ethereum network (Luu et al. 2016).

LOOK AT LUU ET AL.2016

Extant methods of vulnerability detection for Solidity. Multiple tools exists for detecting vulnerabilities in Solidity (Manticore 2018; Mythril 2017; Luu et al. 2016). The most robust and popular of these tools use symbolic analysis. In order to detect whether a program can enter a possible erroneous state, symbolic analysis requires an assertion to

be added to the program which captures that state. A SAT-solver solves for possible values of the variables in that assertion and identifies inputs. This helps determine whether it is possible for a program to ever satisfy the assertion, and if it is, it provides specific test cases for it. Symbolic analysis is the state of the art for preemptive detection of a wide class of bugs (Cadar and Sen 2013). The downside is that it requires assertions to be crafted. Hence, it scales poorly in that it requires experts.

3 Approach

Detecting Vulnerabilities - Ground Truth. Labeling source codes is challenging because, unlike tasks related to images, sound, and simple text, comprehending source codes and and finding nuanced security vulnerabilities is not a task which is inherently easy for a human being to perform. Rather than use trained experts, we proceeded with a scalable, automated way to acquire labels, at the risk of accepting some noise. We ran our codes on state of the art vulnerability detection tools for Solidity, and used their error reports as labels for our modeling task. This provides us a robust way to analyze the strength of our approach.

Program Representation. Building predictive models on programs critically depends on explanatory, independent variables which correlate with the labels, in this case vulnerabilities. Information encoded in such predictive variables can stem from various aspects of a source code. From characters appearing in the raw text of the source code, to values which the program's variables take when tested on different inputs, each provides unique information about the program's functionality. For instance, if a task is to automatically find clones of a snippet of code in a repository, a representation capturing the programs' *structure* would be more important than one which captured token-level information like variable names or operators present in it. Likewise, if the task were to auto-complete code, a *bag of tokens* would be an appropriate representation choice to train models on. We explore which representations might best detect a broad class of vulnerabilities.

We now introduce some nomenclature. Every variable in a program can either be *defined* in a statement - it is initialized or assigned with a value which overwrites its previous value, *used* in a statement - uses the value stored in it in an expression, or not appear in it at all. A *data-dependency* exists between two variables if the value of one variable affects the value in the other. A *control-dependency* exists between a variable and a program construct if the construct can affect the values of the variable. These constructs are generally loops and conditions. We hypothesize that a combination of information present in tokens and the control and data dependencies of programs is required to detect vulnerabilities present in them. Control dependencies provide an insight into the dependence of variables with the structure of the program they appear in. Data dependencies inform how variables exchange information in the scope of a program. This combination can provide an accurate description of the overall functionality of a program, and can be extended to ascertain any general class of vulnerabilities.

Label ($n = 66,470$)	Vulnerable	Not vulnerable
`TOD`	2,360 (4%)	64,110 (96%)
`IntUn`	4,409 (7%)	62,061 (93%)
`IntOv`	18,544 (29%)	47,928 (71%)
`StateChange`	1,266 (2%)	65,204 (98%)

Table 1: Labels summary - `Mythril`. These vulnerabilities have been described in Section 2

We illustrate this intuition through two examples. In Program 2, a dependency on transaction ordering exists because `owner` is *defined* twice, once during the contract initialization and the other in `onlyOwner`, and is *used* in `vote9()`. The order of invoking `onlyOwner` and `vote9()` determines what value of `owner` is set in `vote9()`. Here, the fact that `owner`, when *used* in function `vote9()`, could have previously been *defined* in two different *contexts* is key to determining a TOD issue.

In the case of integer overflows and underflows, just about any *unguarded* addition or subtraction constitutes a vulnerability. We illustrate integer underflows here; overflows behave similarly. A *guarded* expression, as shown in Program 4, is one where the variable to be subtracted (`b`) is checked for certain properties before the subtraction. If most programs use this pattern to check for underflows, then a vulnerability is *signaled* when a variable being subtracted is not used in a condition-check just before.

4 Method

Data Collection Process

To analyze programs written in Solidity, we scraped codes dating from 2015 to 2018, publicly available on `https://etherscan.io`. As of May 2018, we scraped $28,052$ *verified* source files - files verified by Etherscan to be source codes corresponding to their byte codes available on the *Ethereum* blockchain. $25,813$ of them were compilable. Among these, we selected only those which had at least two transactions recorded on *Ethereum*. This served as a proxy for filtering contracts involved in genuine transactions. We were left with $19,023$ files. In total, these files contained $69,599$ contracts, and a total of $487,873$ functions. In our modeling process, code present in each function was treated as an input to the model.

Labeling Process

We surveyed the following tools, all of which have been designed to analyze *Ethereum* smart contracts - `Mythril` (Mythril 2017), `Zeus` (Kalra et al. 2018), `Oyente` (Luu et al. 2016), `Manticore` (Manticore 2018), `Solgraph` (Solgraph 2016), `Solium` (Solium 2018) and `Smart Check` (SmartCheck 2018). We used the following criteria to select a tool - 1. should be based on sound technology and detect non-trivial issues 2. should be open-source, popular, and preferably well maintained. 3. should be able to detect issues in smart-contracts without having to modify/adding additional conditions, `asserts`, *etc*.

Only Mythril and Oyente, two popular and recent tools, both of which use symbolic analysis to predict a variety of issues, matched these requirements. Mythril detects a total of 13 kinds of vulnerabilities, while Oyente detects 8. Only three are common to both tools. We analyzed the consensus between the two tools by running them on our dataset. We found them to agree on roughly 90% of their outputs. Since Mythril produces more granular information, and covers a wider range of issues, we went ahead with it as the sole source to generate our labels. Of the 13 vulnerability-types detected by Mythril, we could work with only 4, since others were either too sparsely represented in our dataset, or had buggy implementations, or reported very general warnings. The distribution of the four vulnerabilities StateChange, TOD, IntUn, and IntOv is provided in Table 1.

Our representation - `Use & Define` paths.

To capture variable usage and definition relationships, we use information similar to what dataflow analysis graphs provide. *use-define*, *define-use*, and other graphs are well established data structures which aid compiler optimizations - they help determine liveness of variables, redundant subexpressions, *etc*. (Aho and Ullman 1977) Such graphs are drawn for every variable in a program $\mathcal{P}$, and each node represents a statement appearing in $\mathcal{P}$. In a *use-define* graph, for instance, for every variable in $\mathcal{P}$, an edge is drawn from a node in which the variable is *used*, to a node in which that value was previously *defined*. In Program 2, variable `owner`, which is used in `balances[owner]` is *defined* in `onlyOwner`.

A Use-Definition Chain (UD Chain) is a data structure that consists of a use, U, of a variable, and all the definitions, D, of that variable that can reach that use without any other intervening definitions. A definition can have many forms, but is generally taken to mean the assignment of some value to a variable (which is different from the use of the term that refers to the language construct involving a data type and allocating storage). A counterpart of a UD Chain is a Definition-Use Chain (DU Chain), which consists of a definition, D, of a variable and all the uses, U, reachable from that definition without any other intervening definitions. Both UD and DU chains are created by using a form of static code analysis known as data flow analysis. Knowing the use-def and def-use chains for a program or subprogram is a prerequisite for many compiler optimizations, including constant propagation and common subexpression elimination.

In our work, we derive similar *use* and *define* information from the Abstract Syntax Tree, $\mathcal{T}$, of programs. Importantly, we capture the control flow-related information along with this *use* and *define* information. For each variable v appearing in a program $\mathcal{P}$, we track four relationships 1. *use-use*: Every *use* of v is tracked to its previous use in $\mathcal{P}$. 2. *use-define*: Every *use* of v is tracked to its previous definition in $\mathcal{P}$. 3. *define-use*: Every *define* of v is tracked to its previous use in $\mathcal{P}$. 4. *define-define*: Every *define* of v is tracked to its previous definition in $\mathcal{P}$. For a variable v, let s_{uv} and s_{dv} correspond to two statements in $\mathcal{P}$ where v is *used* and is previously *defined*, respectively. In tracking the *use-define* relationship, we simply list all the non-terminal nodes in $\mathcal{T}$ that appear in the path connecting s_{uv} and s_{dv} as a feature. By omitting the terminal nodes, we avoid including the specific names of variables. For a given $\mathcal{P}$, our feature space is formed by the distribution of such paths. We similarly generate paths for the other three relationships listed above. In essence, we capture the number of times each such different path appears in $\mathcal{P}$. We correlate these counts to whether a vulnerability exists or not. We discuss this further with an example.

Example. In the sample program described in Figure 2, `b` gets conditionally updated to `b-a` if its value does not equal 10. The AST corresponding to this program is illustrated in Figure 2. In $\mathcal{S}$: $b = b - a$, values of both, `a` and `b`, are *used* to *define* `b`. The *use-define* path of variable `a` is marked in blue. It is defined in the statement `a=5` and is used at

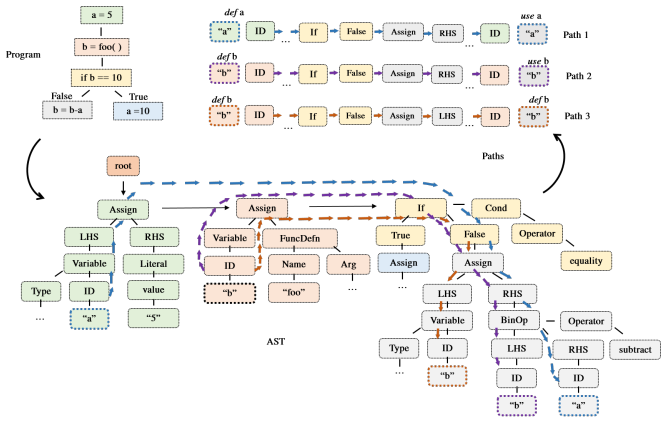

Figure 2: `Use & Define` Paths - An example. Paths to the expression $b = b - a$ are shown. Statements and their corresponding sub-trees have been shaded in the same color.

`b=b-a`. The path (Path1) hence contains ID→Var→LHS→Assign→Assign→If→False→Assign→RHS→BinOp→LHS→ID. Similarly, the *use-define* path of `b` is marked in purple (Path2), and the *define-define* path of `b` is marked in orange (Path3).

The key aspect of these paths is that they fully capture the *control-context* in which these uses and defines happen. For instance, the fact that $\mathcal{S}$ is within the *else* branch of an `If` is captured by the nodes If→False→Assignment→LHS→. . .→Variable in paths involving $\mathcal{S}$. This helps describe both, control and data dependencies of variables in a statement on the rest of the program. We hypothesize that these counts help differentiate structures necessary to semantically define a program. Their counts then serve as a proxy to train a classifier.

Hence, given an AST $\mathcal{T}$ corresponding to a program $\mathcal{P}$, we gather a distribution of such paths. Our feature space thus comprises counts of different path of nodes that describe the four *use* and *define* relationships, for every variable that appears in $\mathcal{P}$. This forms a discrete feature space, and the number of dimensions is the number of such different paths that appear across all $\mathcal{P}$ in the training set. In essence, we form a *count-hot* encoding of these paths as the input signal to our machine learning system.

You put number corresponding to each class in a colection of categrical data type

5 Experiments

In this work, we investigate the following research questions

- How well can a data-driven approach detect vulnerabilities in a Solidity.
- How does a complex program representation influence vulnerability detection?
- Do features selected by models trained on complex representations provide a rich qualitative understanding of the vulnerabilities?

We built a balanced set of roughly 66k functions and used vulnerabilities detected by Mythril on them as our labels. We record path-count based features capturing *use* and *define* information. We trained supervised, binary classification models for each unique vulnerability type detected by

Label	N	Tr	Node	T	UD	UD+T	U	U+T	U+D+UD	U+D+UD+T
Tot # features		2	44	79	8,232	8,311	14,940	15,019	23,172	23,251
Avg # features trained on		2	30	51	1,012	1,161	1,763	1,704	2,775	2,813
TOD	4,594	0.47	0.72	0.70	0.78	0.80	0.76	0.79	0.80	0.80
StateChange	2,440	0.62	0.75	0.75	0.75	0.77	0.77	0.79	0.79	0.82
IntOv	28,634	0.66	0.78	0.81	0.82	0.84	0.84	0.86	0.85	0.87
IntUn	6,676	0.64	0.79	0.85	0.85	0.86	0.85	0.87	0.87	0.88

Table 2: Results summary. Test-set classification accuracies for ablation models.

Mythril. We did not train one model to classify all vulnerabilities to be fair in our comparison with Mythril, which encodes different assertions for each vulnerability. We trained most classes of classifiers available on Scikit-learn (Pedregosa, Varoquaux, and Gramfort 2011). For the best-model, we evaluated results over all such classifiers, and found Logistic regression (L2 penalty, Liblinear solver, Dual formulation, C=1) to consistently perform the best. For brevity, we discuss only Logistic Regression's results.

The dataset pertaining to each label was split into a training and test set in the ratio 66%-33%. The dataset for each label was balanced - with roughly the same number of positive and negative samples, and stratified to ensure that the distribution of function types was similar in the train and test set. Each classification model was trained with three-fold cross-validation. Results were evaluated on the held-out test-set for a given label.

To answer whether our program representations help predict vulnerabilities, we study the following feature classes and perform an ablation study. The number of unique features appearing in the train-set for each of these classes is mentioned in Table 2.

- **Tree-based features, Tr.** We use two simple features - the AST height of the input function, and the number of nodes in the AST. Such high level features pertaining to codes have been used to signal bugs in codes. This serves as a very weak baseline.
- **Node features, Node.** We use the distribution (discrete counts) of the AST nodes of the input function as features. This does not include any concrete values captured in the nodes. For example, a BinaryOp AST node can have a $+$ or a $\times$ present in it. These are not captured.
- **Token features, T.** The distribution of AST nodes along with the concrete values in the nodes are captured. This includes values of operators, type information, string literals *etc*. that are present in the AST nodes. Although we do not present results here, T features produced better results over lexicalized tokens obtained from raw source codes. We hence use the T features as a stronger baseline.
- ***use* and *define* paths, UD, U, D, DU.** The discrete distribution of paths generated by the *use* and *define* relationships are used as features.
- **Combination of features.** We evaluate various combinations of the above models, and denote it with a $+$ symbol.

For models trained on our representations, we also analyze the highest contributing features. Since these are linear models and the features are highly interpretable path information, we gain an insight into what signals the presence of vulnerabilities. These two analyses help answer the three research questions we pose.

"ablation study" has been adopted to describe a procedure where certain parts of the network are removed, in order to gain a better understanding of the network's behaviour.

n the context of the example in the OP - linear regression - an ablation study does not make sense, because all that can be "removed" from a linear regression model are some of the predictors. Doing this in a "principled" fashion is simply a reverse stepwise selection procedure, which is generally frowned upon

6 Results

The test-set classification accuracies of the different models we evaluated are tabulated in Table 2. Each row corresponds to a vulnerability type (labels), while each column denotes the feature class we evaluated on. N corresponds to the total number of functions available for each label. The total number of features available for each model combination are listed in the second row from the top. Feature selection reduces these numbers; the average number of features used per model is mentioned in the third row.

We see that path based information makes for a good representation to capture how programs are written, and encapsulate semantic information about it. We ablate and evaluate different path representations and find them to predict bugs better than a bag of rich tokens. A qualitative analysis of the models trained on these path-features shows features that relate to the overall structure of the vulnerability get automatically discovered and are used to discriminate the presence of a vulnerability.

Discussion. Tr features perform as poorly as a random baseline of 50% accuracy. Node features perform slightly better, suggesting that just the different node types appearing in an AST shows some discrimination to vulnerable codes. We ablate concrete values of operators, types, string literals *etc*. from this model to study their specific contribution. Models trained on T, which include all such concrete values, show an improvement only for the labels IntOv, and IntUn. This is expected since they are informed by the presence of $+$ and $-$ operators.

UD, U vs T. Models trained on the UD and U features see an improvement across all labels. In particular, we see an improvement of 8% points on the test-accuracy on TOD features. This is expected, since TOD is a non-trivial error to describe using just tokens. To maintain brevity, we do not include results for D, DU paths and other ablations involving it. Their results were similar to UD and U.

U+D+UD. Combining the UD and U features (column U+D+UD) provides the best gain in test-accuracy. Models across all labels improve by an average of 7 percentage points. This is a positive result as it shows representations capturing data and control dependencies discriminate vulnerable code-structures. It also validates that these vulnerabilities can be described well in terms of dependencies on other program elements.

Best model. Combining token features `T` along with `UD`, `U`, and `D` features (column `U+D+UD+T`) provides the best model across all labels. We see a statistically significant improvement of an average of 7% per label when compared to the baseline, and a 2% improvement when compared to the `U+D+UD` model. Other metrics such as Precision, Recall, and the F1-score (not included for brevity) also consistently improve across all labels for the combined model. We learn that both, token and path information, independently predict a program's characteristic properties.

Feature importance

For the four vulnerabilities we study, we analyze the top most correlated features from models trained on the two classes - `U+D+UD` and `T`. Due to their similarity, we discuss features of `IntUn` and `IntOv` together. Against each feature, we tabulate its correlation with the label. We list both, positive and negative correlations, since negative ones inform features signaling a lack of vulnerability.

`StateChange.` Table 3 lists features of *State change after external call* vulnerability. Among the token features, function-calls (Fn), modifier-invocation (MI), Emit-statements (EmS), positively correlate with the vulnerability. MI statements are typically included against functions to specify who can access them. These are most frequently used to protect functions where currency is transferred. For example, in Program 3, `onlyOwner`, specified alongside `ownerSafeWithdrawal()`, is an MI which specifies that the function can be accessed only by the contract owner. The prominence of this feature suggests that code-writers are careful enough to place MI around functions which make external calls, but are not aware of this peculiar vulnerability of state-change. Similarly, `emit` is a keyword designed to log transaction, like transferring currency. This feature's prominence suggests that the model learns a correlation between logging of important transactions, presence of function external function calls, and state change vulnerabilities.

In the `U+D+UD` class, we find R→FnD→VD→Fn→ALHS to be prominent. This is a *define-define* feature (see Program 1) and is interpreted as "a variable, declared at the root (R), is defined in an assignment statement (ALHS), where the connecting path contains a function definition (`close`), a variable declaration (`tokens`), and a function call (`transfer`)". This is a standard example of state change after an external call. In case `transfer` hangs, `isClose` can never be set to `True`.

Another prominent feature is VD→B→Fn→EmS→Fn. This is a *use-define* relationship and is interpreted as - "a declared variable (VD) is used in a function call (Fn), and the path connecting these statements contains a block (B), a function call (Fn), and an emit statement (EmS)". The variable `balanceToSend` (marked in green in Program 3) is one whose usage in `ownerSafeWithdrawal` fits this description. This feature automatically discovers this vulnerability as an external function call (`transfer`) being sandwiched between a variable definition (`balanceToSend`) and its use in an `emit`. While this predicts `Mythril`'s output correctly, we observe that this function call is within an `emit`, which is used only for internal logging. This should hence not be of serious concern. We believe this is too severe a warning, and have contacted the authors of `Mythril` and recommended to reduce the severity of their warning in such cases.

Vuln.	T	Correl	U+D+UD	Correl
`StateChange`	MI	+0.26	R→FnD→VD→Fn →ALHS	+0.30
	EmS	+0.20	VD→B→Fn→EmS→Fn	+0.28
	Fn	+0.20	I→Ind →Ind	-0.22
`IntUn, IntOv`	+=	+0.50	ALHS→ES→A→Fn→MA→Ind	-0.32
	−=	+0.42	R→FnD→B→ES→Fn→BiOp	+0.31
	&&	+0.34	Fn→FnD→BiOp	+0.30
`TOD`	Fn	+0.17	ALHS→ES→Fn	+0.32
	Re	-0.18	ALHS→ES→A→Ind	-0.28
	-=	-0.29	VD→FD→ES→Ind	+0.29
			VD→FD→A	+0.26

Table 3: Most represented features in `StateChange`, `IntUn`, `IntOv` and `TOD`

The other prominent `U+D+UD` feature which negatively correlates seems to represent a generic relationship, where a variable (I) is used in an array's index access (Ind) in the scope of the affected function. This seems to be an artefact of the data, and perhaps with a combination of other features, correlates to a lack of vulnerability.

`TOD.` Token features do not provide any meaningful information on transaction ordering dependence vulnerabilities (Table 3). The most significant among them are the number of times a `return` statement occurs, the number of function calls made, and the count of −= appearing in a function. They clearly do not convey any meaningful information regarding the vulnerability. The most prominent `U+D+UD` features are ALHS→ES→Fn, and ALHS→ES→A→Ind. Both signify the appearance of common patterns, namely - "a variable assigned with a value (ALHS) is used in a function call (Fn)", and "a variable assigned (ALHS) is used as an array index (Ind) in an assignment (A)". These are common patterns which the data correlates to implementations involved in transaction ordering bugs. However, the feature VD→FD→ES→Ind, and, VD→FD→A capture a common TOD pattern. The former is a *use-define* relationship, which translates to a variable defined being used as an array index in an expression within a function definition. The latter is a *define-define* relationship which translates to a variable defined being assigned again within a function definition. The co-occurrence of these features thus captures a popular manifestation of a TOD bug appearing in our dataset. Program 2 shows that the order of invoking `onlyOwner` and `vote9()` determines the value of `owner` (marked in orange) in `balances[owner]`.

`IntUn, IntOv.` The most significant token features signaling integer underflows and overflows are the addition and subtraction operators used in an assignment (Table 3). While these are enough to predict 81% and 85% of the cases in the test-set (Table 2), they grossly simplify the problem. The two most prominent `U+D+UD` features represent richer structure. R→FnD→B→ES→Fn→BiOp corresponds to a "variable being assigned in a contract (at the root, R), and being used in a binary operation (BiOp) within a function". This corresponds

```
// is contract close and ended
    bool internal isClose = false;
function close() onlyOwner public {
        // send remaining tokens back to owner.
        uint256 tokens = token.balanceOf(this);
        token.transfer(owner , tokens);
        // withdraw funds
        withdraw();
        // mark the flag to indicate closure of the contract
        isClose = true;
    }
```

Program 1: Correctly predicted `StateChange` vulnerability

```
address public owner = 0xabcabc;
external onlyOwner {
    uint256 balance = balances[owner];
    balances[_newOwner] += balance;
    balances[owner] = 0;
    Transfer(owner, _newOwner, balance);
    owner =  newOwner;
    Owner(_newOwner);
}
function vote9(address _voter, address _votee) external {
    balances[_voter] -= 10;
    balances[owner] += 1;
    balances[_votee] += 9;
    Transfer(_voter, owner, 1);
}
```

Program 2: Correctly predicted `TOD` vulnerability

```
function ownerSafeWithdrawal() external onlyOwner nonReentrant {
    require(fundingGoalReached);
    uint balanceToSend = address(this).balance;
    beneficiary.transfer(balanceToSend);
    emit FundTransfer(beneficiary, balanceToSend, false);
}
```

Program 3: Potential `StateChange` false positive detected by our features

```
function sub(uint256 a, uint256 b) public pure returns (uint256) {
   assert(b <= a);
   return a - b;
  }

function PrivateSaleBuy(address _referrer) returns (bool){
    /*calculate net revenue*/
    ta.n10 = sub(msg.value, ta.n9);
}
```

Program 4: `IntUn` false positive detected by our method

to variable being used in any binary (arithmetic) operation, appearing in a function. This is typical of most simple additions and subtractions taking place in the dataset. The other, which is negatively correlated, suggests "a variable defined in an assignment, being accessed as an array index in an object's member-function call". This can be variable `i` in an expression `obj.add(balanceOf[i])`. This is a structure typical of using safe arithmetic operations in Solidity. The other prominent feature is a *use-use* relation, Fn→FnD→BiOp, which translates to "a variable used in a function call, is used again in the binary operation appearing in another function body". This refers to the structure shown in Program 4 (highlighted in violet). However, it is positively correlated, implying that this structure is prevalent in vulnerable implementations. On studying this feature, we found that `Mythril` systematically flags operations guarded by an `assert` as a vulnerability. This was recently acknowledged by their developers[1], and is in the process of being fixed.

7 Related Work

Recent work have focused on detecting and classifying vulnerabilities through traditional program analysis techniques (Yu, Alkhalaf, and Bultan 2009),(Newsome and Song 2005),(Song et al. 2008), while others focus on efficient and meaningful representations of programs for modeling and prediction in software engineering applications (Allamanis, Peng, and Sutton 2016)(Raychev, Vechev, and Yahav 2014),(Singh, Srikant, and Aggarwal 2016). However, few have worked at the intersection of the two. Our work is closest to unpublished (available on arXiv) work by (Russell et al. 2018). They also use static analyzers for ground truth in C, C++ programs. However, they train a CNN on a bag of lexicalized tokens and then use a Random Forest classifier, which requires large amounts of data. Our work instead focuses on the design of highly interpretable features which capture variable interactions. Our design works in a low-data setting which DSLs like Solidity present.

Work regarding program representations close to ours are (Srikant and Aggarwal 2014), (Alon et al. 2018). (Singh, Srikant, and Aggarwal 2016) introduce features derived from *use-define* paths of programs extracted from ASTs. They consider the two terminal nodes of such paths, and use the control and data-context of those nodes. They do not capture long-range dependencies among variables, spanning multiple functions and classes. We introduce a more general representation for long range dependencies, comprising *usages* and *defines* of variables. Moreover, real-world programs are noisier than restricted student submissions. (Alon et al. 2018) use all valid paths between variables appearing in the AST and train these *bag of paths* using an RNN, again requiring large amounts of data.

8 Conclusion

This work presents an important first step in detecting vulnerabilities for domain specific languages and analyzing programs written in Solidity. The semantic-rich features we introduced capture intricate control and data dependencies and successfully classify four vulnerabilities. Information from program tokens, although semantically incapable of capturing these vulnerabilities, increases accuracy of models. The interpretability of our features allows us to discover new, subtle information about vulnerabilities.

[1] See `https://github.com/ConsenSys/mythril/issues/417`

References

Aho, A. V., and Ullman, J. D. 1977. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc.

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A convolutional attention network for extreme summarization of

source code. In *International Conference on Machine Learning*, 2091–2100.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2018. A general path-based representation for predicting program properties. *arXiv preprint arXiv:1803.09544*.

Cadar, C., and Sen, K. 2013. Symbolic execution for software testing: three decades later. *Communications of the ACM* 56(2):82–90.

Kalra, S.; Goel, S.; Dhawan, M.; and Sharma, S. 2018. Zeus: Analyzing safety of smart contracts. NDSS.

Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; and Hobor, A. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 254–269. ACM.

Manticore. 2018. `https://github.com/trailofbits/manticore`.

Mythril. 2017. `https://github.com/ConsenSys/mythril`.

Newsome, J., and Song, D. X. 2005. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, 3–4. Citeseer.

Pedregosa, F.; Varoquaux, G.; and Gramfort, A. e. a. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.

Raychev, V.; Vechev, M.; and Yahav, E. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, 419–428. ACM.

Russell, R. L.; Kim, L.; Hamilton, L. H.; Lazovich, T.; Harer, J. A.; Ozdemir, O.; Ellingwood, P. M.; and McConley, M. W. 2018. Automated vulnerability detection in source code using deep representation learning. *arXiv preprint arXiv:1807.04320*.

Singh, G.; Srikant, S.; and Aggarwal, V. 2016. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 263–272. ACM.

SmartCheck. 2018. `https://tool.smartdec.net/`.

Solgraph. 2016. `https://github.com/raineorshine/solgraph`.

Solium. 2018. `https://github.com/duaraghav8/Solium`.

Song, D.; Brumley, D.; Yin, H.; Caballero, J.; Jager, I.; Kang, M. G.; Liang, Z.; Newsome, J.; Poosankam, P.; and Saxena, P. 2008. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, 1–25. Springer.

Srikant, S., and Aggarwal, V. 2014. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1887–1896. ACM.

Tzermias, Z.; Sykiotakis, G.; Polychronakis, M.; and Markatos, E. P. 2011. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, 4. ACM.

Wong, W. E.; Gao, R.; Li, Y.; Abreu, R.; and Wotawa, F. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42(8):707–740.

Yu, F.; Alkhalaf, M.; and Bultan, T. 2009. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proceedings of the 2009 IEEE/ACM International Conference on automated software engineering*, 605–609. IEEE Computer Society.