Demystifying Loops in Smart Contracts

Benjamin Mariano
bmariano@cs.utexas.edu
University of Texas at Austin

Yanju Chen
yanju@cs.ucsb.edu
University of California, Santa Barbara

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara

Shuvendu Lahiri
shuvendu.lahiri@microsoft.com
Microsoft Research

Isil Dillig
isil@cs.utexas.edu
University of Texas at Austin

ABSTRACT
This paper aims to shed light on how loops are used in smart contracts. Towards this goal, we study various syntactic and semantic characteristics of loops used in over 20,000 Solidity contracts deployed on the Ethereum blockchain, with the goal of informing future research on program analysis for smart contracts. Based on our findings, we propose a small domain-specific language (DSL) that can be used to summarize common looping patterns in Solidity. To evaluate what percentage of smart contract loops can be expressed in our proposed DSL, we also design and implement a program synthesis toolchain called Solis that can synthesize loop summaries in our DSL. Our evaluation shows that at least 56% of the analyzed loops can be summarized in our DSL, and 81% of these summaries are exactly equivalent to the original loop.

ACM Reference Format:

1 INTRODUCTION
As programs that run on a blockchain, smart contracts enable multi-party transactions that involve the transfer of funds or other commodities. Due to their obvious security-critical nature, there has been a flurry of interest — both in the security, software engineering, and formal verification communities — in developing techniques for automatically checking that smart contracts behave as expected. For example, some of these techniques look for certain patterns, such as re-entrancy, that are highly correlated with security vulnerabilities in practice [20, 34]. Other proposals aim for full functional correctness and propose verification tools for checking smart contracts against manually-written formal specifications [27, 30, 38, 40].

Despite numerous differences between existing contract analysis techniques, a common unifying theme behind these techniques is the assumption that loops occur rarely in smart contracts [40]. As a result, most existing program analysis techniques in this space brush aside the classical problem of loop invariant generation and focus on challenges that are unique to smart contracts.

The goal of this paper is to study how loops are used in smart contracts and examine how frequently they occur in practice. Our main motivation behind this study is to guide and inform future research on analysis techniques for smart contracts. Specifically, we focus on Solidity, the most popular programming language for smart contract development, and study over 20,000 Solidity contracts deployed on the Ethereum blockchain. In addition to evaluating loop frequency, we also investigate several interesting syntactic and semantic properties of smart contract loops. Our primary goal in this study is to understand the nature of loops that occur in smart contracts so that subsequent research can develop suitable abstract domains and other analysis techniques for more precise static reasoning about smart contracts.

As a first step towards achieving this goal, we implement a basic static analysis to extract several interesting features of smart contract loops. We then use the results of this analysis to cluster semantically similar loops and manually study the behavior of different clusters. Our manual investigation suggests that most smart contract loops can be summarized using a small domain-specific language (DSL) with familiar functional operators, such as map, fold, and zip.

As a next step towards precise reasoning about smart contract loops, we build a program synthesis toolchain for generating summaries of Solidity loops in our proposed DSL. Our synthesizer performs type-directed search over the DSL constructs and uses properties of the loop body to significantly prune the search space of candidate summaries. Once we generate a candidate summary, we then check for equivalence against the original Solidity loop using a (bounded) equivalence checker built on top of Rosette [44].

We have implemented our smart contract loop summarization technique in a tool called Solis and evaluate it on a large collection of Solidity loops. Our evaluation suggests that a majority of smart contract loops can be summarized (either partially or precisely) in our proposed DSL by leveraging program synthesis.

In summary, this paper makes the following contributions:

- We perform an empirical study of loops in over 20,000 Solidity contracts deployed on the Ethereum blockchain and study various syntactic and semantic properties of these loops. Our study shows that half of the contracts with over 200 lines of code contain at least one loop and that a typical contract contains one loop per 250 lines of Solidity code.
• We cluster smart contract loops according to their semantic features and manually investigate each cluster to understand common behaviors of Solidity loops.
• We use the results from our manual investigation to design a domain-specific language called Consul for summarizing smart contract loops using familiar functional operators.
• We implement a program synthesis toolchain called Solis for summarizing solidity loops in Consul, which uses type information and properties of the input loop to reduce the space of candidate loop summaries that it needs to consider.
• We perform an evaluation of our synthesizer on 1220 loops and show that almost half of the loops can be precisely summarized in our DSL using the proposed synthesis technique.

2 A STUDY OF LOOPS IN SMART CONTRACTS

In this section, we provide statistics about the frequency of loops in smart contracts and report on the distribution of loops according to various features. To perform this study, we collected over 27,000 smart contracts from Etherscan and used the Slither static analyzer [12] to analyze their loops. In this section, we summarize our findings for the 22,685 contracts that could be analyzed by Slither.\footnote{Slither is unable to analyze some of these contracts due to unsupported versions or features of Solidity.}

2.1 Frequency of Loops

Since many program analyzers for smart contracts are based on the assumption that loops are rare, we first study the frequency of loops in smart contracts written in Solidity.

Figure 1 presents a bar chart showing the percentage of contracts that contain at least one loop (in red). Specifically, each bar corresponds to a family of smart contracts grouped according to their lines of Solidity code. Across all contracts, we find that roughly one in five contracts have at least one loop, suggesting that loops may not be quite as rare as assumed in prior work [40]. However, the percentage of contracts that contain loops increases significantly as we consider larger contracts. In particular, for contracts between 500-1000 lines of code, roughly four in five contracts contain at least one loop, and almost no contract with over 1000 lines of code is loop-free.

Next, we study the frequency of loops within a typical smart contract. The blue bars in Figure 1 show the frequency of loops per hundred lines of source code, again grouped according to lines of Solidity code. Across all contracts, we encounter an average of one loop per 250 lines of source code. However, this ratio is significantly higher for larger contracts. In particular, for contracts with over 500 lines of code, we encounter an average of one loop per 100-125 lines of source code.

**Key finding 1:** Roughly one in five contracts contain at least one loop, and only half of the contracts with over 200 lines of code are loop-free. Across all contracts, we find an average of one loop per every 250 lines of source code.

2.2 Nature of Loops, Syntactically

In this section, we investigate various syntactic properties of loops in smart contracts. In particular, Table 1 presents information about the size of loops for a family of contracts grouped according to their size. Across all contracts, we find that the median size of the loop body is two lines of code, and, unlike frequency (Figure 1), we see that loop size does not seem to vary much according to contract size. As also shown in Table 1, it is very uncommon to see nested loops in smart contracts: The median nesting level, including for contracts over 1000 lines of code, is just one (i.e., no nested loops).

Furthermore, a typical loop touches an average of roughly five
program variables.

Key finding 2: A typical smart contract loop is quite small, containing roughly two lines of code in their body. Furthermore, nested loops are very uncommon.

Next, as summarized in Table 2 we study what percentage loops exhibit various syntactic properties. Our first finding is that the overwhelming majority of smart contracts (around 88%) involve collections, meaning that they iterate over elements of arrays or mappings. Furthermore, roughly three out of four loops perform writes, meaning that they involve side effects on contract state. Another interesting observation is that, while the average loop body contains approximately two lines of code, more than one in three loops contain at least one control flow construct (e.g., if statement, continue etc.) in their body. Other salient findings from our study can be summarized as follows:

- **Batch transfer:** Loops that involve calls to built-in Solidity functions like send and transfer typically perform so-called batch transfer, meaning that they transfer ether to a set of recipients. According to our findings, 8.5% of Solidity loops perform batch transfer.
- **Enforcing requirements:** Programmers use the require construct in Solidity to establish some pre-condition. We find that 13% of loops involve a require statement, suggesting that these loops establish some universally quantified pre-condition involving a data structure.
- **Nested loops:** As we saw earlier, nested loops are uncommon, and only 4.5% of loops contain a nested loop.
- **SafeMath:** Approximately 13% of the loops involve a call to the SafeMath library for avoiding overflows.
- **Events:** Solidity programs can emit so-called events that facilitate the convenient usage of EVM logging facilities. We find that the emission of events is quite uncommon within loops – only 4.5% of loops involve events.
- **Function calls:** Roughly one in five Solidity loops contain a call to a function (excluding saemath calls and built-in functions like transfer).
- **Bounded loops:** Approximately 18% of the loops have a constant bound, while the remaining ones all have symbolic bounds.

Key finding 3: The overwhelming majority of loops iterate over collections and modify contract state. However, since function calls within loops are not very common, analysis of most loops does not require interprocedural reasoning. Furthermore, 80% of Solidity loops have symbolic bounds; therefore, analysis techniques based on loop unrolling cannot soundly handle most smart contract loops.

2.3 Nature of Loops, Semantically

In this section, we investigate some interesting semantic properties of smart contract loops. To perform this study, we use the Slither infrastructure [12] to implement a basic static analysis for analyzing data dependencies between variables. The results of this analysis are summarized in Table 3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops that involve collections</td>
<td>87.58%</td>
</tr>
<tr>
<td>Loops that involve writes</td>
<td>76.27%</td>
</tr>
<tr>
<td>Loops that involve events</td>
<td>4.50%</td>
</tr>
<tr>
<td>Loops that have function calls</td>
<td>21.95%</td>
</tr>
<tr>
<td>Loops that have control flow constructs</td>
<td>37.39%</td>
</tr>
<tr>
<td>Loops that have nested loops</td>
<td>4.52%</td>
</tr>
<tr>
<td>Loops that have transfers</td>
<td>8.46%</td>
</tr>
<tr>
<td>Loops that have requires</td>
<td>12.98%</td>
</tr>
<tr>
<td>Loops involving SafeMath</td>
<td>13.25%</td>
</tr>
<tr>
<td>Loops that have a constant bound</td>
<td>17.79%</td>
</tr>
</tbody>
</table>

Table 2: Syntactic properties of loops

<table>
<thead>
<tr>
<th>Type of dependence</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar value depends on collection</td>
<td>18.14%</td>
</tr>
<tr>
<td>Collection depends on collection</td>
<td>23.77%</td>
</tr>
<tr>
<td>Collection depends on scalar</td>
<td>15.90%</td>
</tr>
<tr>
<td>Collection key depends on collection value</td>
<td>14.84%</td>
</tr>
</tbody>
</table>

Table 3: Semantic properties of loops

Our first finding is that 18% of the smart contract loops involve some data dependency from collections to scalars. Such data dependencies typically arise when the loop performs some computation by iterating over a data structure, such as computing the sum of all elements in an array.

Next, we find that 24% of loops involve data dependency between a pair of collections (including dependencies between different elements of the same collection). For example, such dependencies arise when shifting array elements or building a new data structure based on values stored in other data structures. On the other hand, dependencies of collections on scalars is slightly less common; roughly 16% of loops exhibit such a dependency. For instance, a loop that sets some range of array elements to a constant value would exhibit such a collection-on-scalar dependency.

Finally, we study how often collection elements are used to index into other collections. For instance, consider a scenario where an array A stores all participants in an auction, and another mapping M from participants to amounts stores the bid of each participant. In this scenario, one may need to loop over all elements in array A and use A[i] to access the bid of a given participant in M. We refer to this type of dependency as value-to-key dependency and find that 15% of loops involve such a pattern.

2.4 Clustering Loops by Semantic Properties

In this section, we identify common looping patterns in smart contracts using a combination of k-means clustering and manual sampling. In particular, we use the scikit-learn Python library [39] to facilitate semantic clustering of Solidity loops, and then manually inspect samples from each cluster to determine common semantic behaviors per cluster.

To perform this clustering, we first reduced the loops to dependency graphs similar to DroidSift [47] and Holmes [18]. As the four properties in Table 3 are clearly not sufficient for clustering,
we broke down these properties into common dependency graph features [18, 47], such as the number of different types of nodes that are being read from or written to. In total, we used 37 features that serve as refinements of the properties from Table 3. For a complete list of the features used, see Appendix A. To identify only common loop patterns, we restricted the number of clusters to 6. Additionally, to reduce overlap in clusters, we limited loops to those with single-statement bodies that contain only common function calls (transfers, requires, and SafeMath functions).

Figure 2 shows the results of our clustering, projected onto three dimensions by performing a high-dimensionality reduction in a scatter plot. For each cluster, we randomly sampled 10 files, and manually inspected each sample. We will now explain some of the salient findings from our manual investigation.

Fold pattern. Both cluster 0 (pictured in red in Figure 2) and cluster 1 (in green) contain loops that perform an accumulating computation similar to a fold in functional programming. In particular, 6 of the 10 sampled loops from cluster 0 sum the values of an array/mapping, while 8 of 10 from cluster 1 perform a fold using a more complicated arithmetic function.

Map pattern. Clusters 2 (in blue), 3 (in orange), and 4 (in purple) contain loops that map values onto a global array/mapping, similar to the map combinator in functional programming. In particular, 8 of the 10 loops sampled from cluster 2 write a constant value \( c \) to mapping \( m \) indexed by an array \( a \) (i.e., \( m[a[i]] = c \) where \( i \) is the loop iterator). 7 of the 10 sampled loops in cluster 3 and 5 of 10 from cluster 4 contain a write from one mapping \( m_1 \) to another \( m_2 \), e.g., \( m_1[i] = m_2[i] \).

Require. In addition to the mapping pattern found in cluster 4 (in purple), we also found that 3 of the 10 sampled loops contain calls to the built-in require function, which reverts execution of the smart contract if a user-defined condition is not met. For all three loops, the user-defined condition was a linear constraint on values of an array/mapping.

Arithmetic. Finally, the loops sampled from cluster 5 (in grey) all contain complex arithmetic computations over global integers. Furthermore, all of these loops use arithmetic functions from the common SafeMath library, which may be why they were clustered together. However, this cluster is significantly smaller than all the other clusters — instances in this cluster only constitute 2% of the loops used for this analysis.

In addition to sampling instances from each cluster, we also sampled larger loops to see if their behavior can be expressed as a combination of the behaviors observed in each of our clusters. While we found many of the same behaviors, our manual investigation of larger loops revealed another operator, namely zipWith, that is useful for characterizing smart contract loops. In particular, loops involving this pattern perform pair-wise aggregation between two arrays/mappings. However, we found several variants of the zipWith pattern, where one or both of the mappings are indexed by another mapping/array.

3 A DSL FOR LOOP SUMMARIZATION

Based on our findings from the previous section, we now propose a small-but-expressive DSL called Consul that can be used to summarize common Solidity loops. We believe this DSL can aid analysis, verification, and optimization efforts for smart contracts by capturing common loop behaviors and facilitating the design of suitable abstract domains for analyzing smart contract loops.

The syntax of the Consul summarization language is shown in Figure 3. At a high level, Consul programs are compositions of assignments \( \phi = \mathcal{E} \) (where \( \phi \) is a program variable), requirements, and ether transfers. Specifically, assignments capture side effects of loops on program variables, and multiple side effects are expressed via composition. In addition to capturing side effects on explicit contract state, our DSL also captures the semantics of loops that involve transfer and require, which occur in a non-trivial portion of loops (recall Table 2).

In the remainder of this section, we give a high-level description of Consul semantics and refer the interested reader to Appendix B for the formal semantics of each construct.

3.1 Summarizing Side Effects on Contract State

Consul programs summarize the side effects of Solidity loops using three functional operators and their variants. All of these constructs operate over mappings and produce either scalars or mappings.

Map operator. The most basic construct in Consul is the map operator \( \text{map}(m, \phi, \mathcal{F}) \) which yields a new mapping \( m' \) which is the same as \( m \) except that any key-value pair \( (k, \phi) \) satisfying \( \phi \)

Appendix is included in the supplementary material.

Initially we tried less restrictive clustering, but this led to clusters whose common behaviors were hard to identify.

3.2, 3.3...
Core constructs

\begin{align*}
\text{Stmt } S & \rightarrow \nu = E \mid S; S \mid \text{transfer}(m_1, m_2, F, \varphi) \\
& \mid \text{require}(m, \varphi_1, \varphi_2) \\
& \mid \text{requireNested}(m_1, m_2, \varphi_1, \varphi_2)
\end{align*}

\begin{align*}
\text{Expr } E & \rightarrow m \in \text{Mappings} \\
& \mid \text{map}(m, \varphi, F) \\
& \mid \text{foldl}(m, \varphi, f, i) \text{ where } f \in \{+, \max, \ldots\} \\
& \mid \text{zip}(m_1, m_2, \varphi, f) \text{ where } f \in \{+, \max, \ldots\} \\
& \mid \text{mapNested}(m_1, m_2, \varphi, F) \\
& \mid \text{foldlNested}(m_1, m_2, \varphi, f, i) \text{ where } f \in \{+, \ldots\} \\
& \mid \text{zipNestedSym}(m_1, m_2, m_3, \varphi, f), \ f \in \{+, \ldots\} \\
& \mid \text{zipNestedASym}(m_1, m_2, m_3, \varphi, f), \ f \in \{+, \ldots\}
\end{align*}

Functions

\begin{align*}
\text{Function } F & \rightarrow \lambda x. t \\
\text{Term } t & \rightarrow x | i \\
& \mid x \oplus i \text{ where } \oplus \in \{+, \times, -, /\} \\
\text{Int } i & \rightarrow c \in \text{Int} \mid e \in \text{IntProgExprs}
\end{align*}

Predicates

\begin{align*}
\text{Predicate } \varphi & \rightarrow \lambda(x,y). \phi \\
\text{Formula } \phi & \rightarrow \text{true} \mid \text{inRange}(x, t_1, t_2) \\
& \mid \nu \times t_1 \text{ where } \nu \in \{x, y\}, \ k \in \{\le, \leq, <, \ge, >\} \\
& \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2
\end{align*}

Figure 3: The \textsc{Consul} DSL for expressing loop summaries

is changed to \((k, F(\nu))\). For instance, a loop that increments the first \(n\) elements in an array \(a\) by one can be summarized using the following \textsc{Consul} statement:

\[ a = \text{map}(a, \lambda(x,y). \text{inRange}(x, 0, n), \lambda x. x + 1) \]

\textit{Foldl operator}. As mentioned in Section 2.3, roughly one in five Solidity loops perform some sort of aggregation over a data structure. To capture this common pattern, \textsc{Consul} provides a construct of the form \text{foldl}(m, \varphi, f, i) that aggregates all values in \(m\) satisfying \(\varphi\) by using function \(f\) and initial value \(i\). As an example, the statement \(x = \text{foldl}(a, \lambda x. x + 0, 0)\) captures the behavior of a loop that computes the sum of all elements in array \(a\) and assigns it to variable \(x\).

\textit{Zip operator}. Another common pattern we observed in Solidity loops is to perform pair-wise aggregation between two mappings. \textsc{Consul} captures this common behavior using the \text{zip} construct. For example, the statement \(c = \text{zip}(a, b, \lambda x. x + 0)\) captures the effect of a loop that constructs a new array \(c\) where each element \(c[i]\) is the sum of \(a[i]\) and \(b[i]\).

\textit{Nested family of operators}. As discussed in Section 2.3, many Solidity loops iterate over a mapping \(m_1\) and use values in \(m_1\) to access values in another mapping \(m_2\). Furthermore, we found this pattern to occur frequently both for performing aggregation as well as building new data structures. Therefore, the \textsc{Consul} language provides a variant of each of the mapping, \text{zip} and \text{foldl} constructs for expressing such behaviors.

Specifically, the construct \text{mapNested}(m_1, m_2, \varphi, F) is similar to map except that lambda expression \(F\) is applied to elements \(m_1[m_2[i]]\) (rather than \(m_1[i]\)) for keys/indices \(i\) satisfying predicate \(\varphi\). For instance, consider a smart contract that has an array called \(a\) that stores a designated set of addresses as well as another array called \(b\) that maps each address to some amount. Now, consider a loop that increments the amount of money for each address in \(a\) by some amount \(amt\). This complex pattern can be summarized concisely using the \text{mapNested} pattern as follows:

\[ b = \text{mapNested}(b, \lambda x. x + \text{amt}) \]

Next, the nested variant of \text{foldl}, called \text{foldlNested} allows performing nested aggregation. In particular, \text{foldlNested}(m_1, m_2, \varphi, f, i) aggregates using function \(f\) all elements \(m_1[m_2[i]]\) for keys/indices \(i\) satisfying predicate \(\varphi\). For instance, the statement

\[ \text{foldlNested}(a, b, \lambda(x,y). \text{inRange}(x, 1, 3), +, 0) \]

computes the sum \(a[b[1]] + a[b[2]] + a[b[3]]\).

Finally, the two \text{zipNested} constructs provide nested variants of \text{zip}. Specifically, consider three mappings:

\[ m_1 : r_1 \Rightarrow r_2 \quad m_2 : r_1 \Rightarrow r_2 \quad m_3 : r_3 \Rightarrow r_1 \]

Then, \text{zipNestedSym}(m_1, m_2, m_3, \varphi, f) creates a new mapping \(m_4\) of type \(r_1 \Rightarrow r_2\) where \(m_4[m_3[i]]\) is equal to \(f(m_1[m_3[i]], m_2[m_3[i]])\) for those indices \(i\) satisfying \(\varphi\).

The other variant \text{zipNestedASym} is similar except that it is asymmetric. Specifically, it operates over mappings with the following types:

\[ m_1 : r_1 \Rightarrow r_2 \quad m_2 : r_3 \Rightarrow r_2 \quad m_3 : r_3 \Rightarrow r_1 \]

Then, the construct \text{zipNestedASym}(m_1, m_2, m_3, \varphi, f) creates a new mapping \(m_4\) of type \(r_1 \Rightarrow r_2\) where:

\[ m_4[m_3[i]] = f(m_1[m_3[i]], m_2[i]) \]

for those indices \(i\) satisfying \(\varphi\). In practice, we found this asymmetric variant of \text{zipNested} to occur more frequently than its symmetric variant.

### 3.2 Summarizing Transfers and Requirements

Recall from Section 2.2 that roughly one in five Solidity loops involve calls to built-in Solidity functions such as \text{require} and \text{transfer}. Thus, our summarization language also provides constructs for capturing the behavior of such loops. For instance, the construct \text{transfer}(m_1, m_2, F, \varphi) is used to summarize batch transfers where \(m_1\) is a mapping \(\tau \Rightarrow \text{Address}\) and \(m_2\) is a mapping \(\tau \Rightarrow \text{Int}\). In particular, \text{transfer}(m_1, m_2, F, \varphi) indicates the batch transfer of amount \(m_2[i]\) from the receiver object to address \(m_1[i]\) for all keys \(i\) satisfying predicate \(\varphi\).

Finally, our DSL provides two variants of require for summarizing loops that enforce some pre-condition. In particular, the construct \text{require}(m, \varphi_1, \varphi_2) checks that predicate \(\varphi_2\) is satisfied for all elements \(m[i]\) where \(i\) satisfies \(\varphi_1\). The \text{requireNested} construct is similar and corresponds to the nested variant of require. Specifically, \text{requireNested}(m_1, m_2, \varphi_1, \varphi_2) checks that predicate \(\varphi_2\) is satisfied for all elements \(m_1[m_2[i]]\) where \(i\) satisfies \(\varphi_1\).


### Table 4: Examples of Solidity loops and their Consul equivalent

<table>
<thead>
<tr>
<th>Solidity Loop</th>
<th>Consul Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>`for(j = 0; j &lt; addrs.length; j++){</td>
<td><code>bal = zipNestedASym(bal, amts, addrs, inRange(x, 0, addrs.length), +)</code></td>
</tr>
<tr>
<td>bal[addrs[j]] += amts[j];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>`for(i = 0; i &lt; _addrs.length; i++){</td>
<td><code>wlst = mapNested(wlst, _addrs, inRange(x, 0, _addrs.length) \land \neg y, true)</code></td>
</tr>
<tr>
<td>if (!_wlst[_addrs[i]]) {</td>
<td></td>
</tr>
<tr>
<td>wlst[_addrs[i]] = true;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>`for(j = 0; j &lt; addrs.length; j++){</td>
<td><code>require(cur != 0x0);</code></td>
</tr>
<tr>
<td>cur = addrs[j];</td>
<td><code>requireNested(uTime, addrs, inRange(x, 0, addrs.length), x &lt; now);</code></td>
</tr>
<tr>
<td>require(now &gt; uTime[cur]);</td>
<td><code>amts = map(amts, inRange(x, 0, addrs.length), x*1e8);</code></td>
</tr>
<tr>
<td>amts[j] *= 1e8;</td>
<td><code>total = foldl(amts, inRange(x, 0, addrs.length), +, total)</code></td>
</tr>
<tr>
<td>total += amts[j];</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Examples

In this section we provide some sample Solidity loops and show how they can be summarized in the Consul DSL.

**Example 3.1.** The first row in Table 4 shows a single line Solidity loop that increments the balance of a list of addresses by a given amount. This loop can be summarized using the asymmetric version of the zipNested construct, as shown on the right-hand-side of Table 4.

**Example 3.2.** The second row of Table 4 shows a loop that whitelists certain addresses if that address is not already on the whitelist. This loop can be summarized using the mapNested construct, as shown in the second column of Table 4. Observe that the predicate ensures that, for any updated element, the key must be in the range \([0, \text{addrs.length}]\) and the corresponding value must be false.

**Example 3.3.** The last row in Table 4 shows a more complex loop that establishes a pair of pre-conditions, while also having side effects on contract state. As shown on the right hand side of the table, the summary of this loop involves the requireNested construct since the requirement on the third line of the loop body involves a nested data structure access. The update to the amts array is captured using the map construct and the computation of total amount is captured using foldl with the + operator.

4 SYNTHESIS OF LOOP SUMMARIES

As a first step towards reasoning about loops in smart contracts, we implemented a tool called Solis for automatically synthesizing loop summaries. Our tool is based on the syntax-guided synthesis paradigm [3] and, as shown in Figure 4, it leverages both an a type-directed search engine and a (bounded) equivalence checker. Specifically, the search engine enumerates candidate Consul summaries for the given loop and uses the verifier to test equivalence between the loop and candidate summary. The search engine continues this enumeration process until we either exhaust all candidate Consul programs for the given loop or until we generate an equivalent summary. In the remainder of this section, we provide a brief overview of the search engine and equivalence checker underlying the Solis summary synthesis engine.

4.1 Type-Directed Search Engine

Similar to several other synthesizers [4, 14, 17], Solis enumerates DSL programs in increasing order of complexity and leverages properties of the input loop to avoid enumerating useless summaries. Specifically, our search engine utilizes both type information as well as basic static analysis of the loop body to prune large parts of the search space.

In more detail, Table 5 presents necessary conditions for our search engine to enumerate a given Consul statement. These pre-conditions make use of two main types of information:

- **Variable types:** Given a variable \(v\), we write \(\tau(v) = \tau_1\) to indicate that the type of variable \(v\) is \(\tau_1\). If \(\tau_1 = \text{Map}\), the notation \(\tau_2 \Rightarrow \tau_3\) indicates that the key type of the map is \(\tau_2\) and value type is \(\tau_3\).
- **Read-write sets:** For each loop \(L\) in the original contract, we compute the set of variables it reads from and the set of variables it writes to. We use the notation \(v \in \text{Reads}(L)\) (resp. \(v \in \text{Writes}(L)\)) to indicate that \(v\) is read from (resp. written to) in loop \(L\).

Since the pre-conditions in Table 5 are fairly self-explanatory, we do not describe all of them in detail but focus on a few example constructs:

- **map:** According to the second row of Table 5, Solis only enumerates the statement \(v_1 = \text{map}(v_2, \varphi, F)\) if both \(v_1\) and \(v_2\) have type mapping and \(v_1\) is written to by the input loop whereas \(v_2\) is read from.
- **foldlNested:** For an input loop \(L\), the summary should only contain the statement \(v_1 = \text{foldlNested}(v_2, v_3, f, i)\) if \(v_1\) has type int and the value type of \(v_2\) is the same as the key type of \(v_2\). In addition, \(v_1\) must be written to and \(v_2, v_3\) must be read from.
- **Composition:** According to the first row of Table 5, different statements in the summary should not write to the same variable. This rule prevents Solis from enumerating useless
For a given Solidity loop \( L \), there are typically not too many \textsc{Consul} statements that meet the pre-conditions from Table 5; thus, our pre-conditions are effective at pruning a large part of the search space during enumeration.

In addition to the rules shown in Table 5, \textsc{Solis} uses several other type-directed rules to prune the space of functions it needs to consider. For example, consider again the statement:

\[
v_1 = \text{map}(o_2, \varphi, F)
\]

Clearly, for this program to type-check, the type of \( t \) must be the same as the value type of \( v_1 \). Thus, \textsc{Solis} can also use type-based reasoning when filling in lambda expressions used in \textsc{Consul} constructs.

### 4.2 Equivalence Checker

Once \textsc{Solis} generates a candidate summary, it needs to check whether the summary is equivalent to the original loop \( L \). Towards this goal, we first convert the \textsc{Consul} summary \( S \) into equivalent Solidity code \( S' \) using syntax-directed translation (see Appendix B). However, since there is no off-the-shelf equivalence checker for Solidity programs, we also implemented a (bounded) verifier for checking equivalence between a pair of Solidity programs.

The key idea underlying our equivalence checker is illustrated in the code shown in Figure 5. Specifically, the checker takes as input two Solidity programs \( \text{fun1} \) and \( \text{fun2} \) as well as a symbolic input called \text{input} and verification bound \( K \) that controls how many times the loops are unrolled. Then, we symbolically evaluate both \( \text{fun1} \) and \( \text{fun2} \) on the symbolic input state to obtain a pair of symbolic output states \( o_1 \) and \( o_2 \). If the resulting output states are equal, this constitutes a proof (up to bound \( K \)) that the \textsc{Consul} summary is equivalent to the original loop.

We have implemented our equivalence checker on top of the \textsc{Rosette} framework [44] and leverage its SMT encoding facilities as well as its symbolic evaluation engine. Specifically, we implemented a translator from Solidity code to an intermediate representation that can be symbolically evaluated by \textsc{Rosette}.

### 4.3 Compositional Synthesis

As is common in program synthesis, the search space of programs we must consider in \textsc{Consul} increases exponentially in the number of statements. To scale our synthesis procedure to larger \textsc{Consul} programs, we introduce \textit{compositional synthesis}, which enables synthesis of complex summaries by synthesizing their constituent parts and then composing them together.

In particular, \textit{compositional synthesis} works by independently synthesizing \textit{partial summaries} for each variable written in the loop, and then composing them together with \textsc{Consul}’s sequencing operator. Consider the following loop, which can be expressed in

![Figure 4: Overview of \textsc{Solis}](image)
Consul as a fold over arr (stored in acc) followed by a map over arr (stored in arr).

```plaintext
for(uint i = 0; i < len; i++){
    acc += arr[i];
    arr[i] *= 5;
}
```

We start by synthesizing a single-statement summary for each variable which is written in the loop; in this case we synthesize the following two summaries:

```plaintext
acc = foldl(arr, inRange(x, 0, len), +, acc)
arr = map(arr, inRange(x, 0, len), x * 5)
```

Finally, we compose together the two summaries using the sequencing operator ; which gives us the complete summary.

While this optimization does not affect the soundness of our synthesis algorithm, it does make it incomplete. In particular, this algorithm will not be able to synthesize a Consul program \( P = S_1; S_2 \) where statement \( S_2 \) depends on statement \( S_1 \) — that is, where variables read in \( S_2 \) are written in \( S_1 \). This is because there is no valid single-statement partial summary which captures \( S_2 \) without \( S_1 \). In practice, we observed relatively few loops with dependent statements, and thus found it worth sacrificing completeness for scalability.

5 EVALUATION

To evaluate what percentage of Solidity loops can be summarized in our proposed Consul language, we used Solis to synthesize summaries for real-world Solidity loops. The main goal of our evaluation is to answer the following research questions:

**RQ1** What percentage of loops can be precisely (or at least partially) summarized in the Consul language?

**RQ2** What is the relative frequency of Consul constructs that are used in Solidity loop summaries?

**RQ3** What types of loops cannot be summarized using Solis?

5.1 Experimental Set-up

We conduct this experiment on approximately 1,200 contracts that contain loops. Specifically, to perform this experiment, we extract all loops contained in these 1,200 contracts and then filter out loops that contain Solidity features that cannot be handled by Solis. Such features include user-defined constructs, multi-dimensional arrays, and calls to functions that cannot be analyzed by Solis. Using this methodology, we obtained a total of 1,220 benchmarks, which corresponds to approximately 40% of all loops found within the original 1,000 Solidity contracts.

All experiments reported in this section are conducted on a t3.2xlarge machine on Microsoft Azure with an Intel Xeon Platinum 8000 CPU and 32G of memory, running the Ubuntu 18.04 operating system and using a timeout of 120 minutes for each loop.

5.2 Key Results

We now summarize our key findings as they relate to the research questions posed earlier.

Table 6 presents the result of running Solis over our 1,220 benchmarks. Here, the first column shows the size of the loop body in terms of lines of Solidity code, and the second column shows the total number of loops of that size. The next column labeled “Time-outs” shows the number of benchmarks for which Solis fails to finish its exploration within the given time limit. Next, the column labeled “Precise sum.” shows the number of loops for which Solis can generate a precise, partial, or both summaries, including those benchmarks where we encounter a time-out. The last column reports the same information but excludes those benchmarks on which Solis times-out. In the remainder of this discussion, we mostly focus on those benchmarks on which Solis does not time out, as our primary goal is to investigate Solidity loops rather than the Solis synthesizer.

As we can see from Table 6, Solis can generate a summary for 65% of the benchmarks, and we find that these summaries are equivalent to the original loop for 53% of the benchmarks on which we do not observe a time-out. As we can also see from Table 6, the percentage of loops that can be summarized by Solis is smaller for larger loops. However, upon manual inspection, we find that this is often due to the incomplete decomposition strategy described in Section 4.3. In particular, while this decomposition allows our synthesis technique to be more scalable, it sacrifices completeness in situations where one side effect of the loop is dependent upon another one. Thus, in reality, the percentage of loops that have a Consul summary is much higher than the data presented in Table 6.

**Result for RQ1**: At least 56% of the loops in Solidity contracts have a summary that is expressible in the Consul loop summarization DSL, and 81% of these summaries are equivalent to the original loop.
Next, to answer our second research question, Figure 6 shows the relative frequency of each CONSUL construct in the generated summaries. For each DSL operator (map, fold etc), we group together its nested and non-nested version — e.g., the bar labeled zip includes all three variants of the zip construct. Finally, the last two columns differentiate between the nested and non-nested version of the DSL operators. For example, the bar labeled “nested” includes mapNested, foldINested, requireNested and the two nested variants of zip.

As we can see from Figure 6, map is the most commonly occurring DSL construct, followed by the require operator. The next two most common operators are foldl and zip, and by far the least common construct is transfer. Another interesting observation from Figure 6 is that the nested versions of the operators occur more frequently than the non-nested versions (57% vs. 43%).

**Result for RQ2**: The most commonly occurring CONSUL construct in the synthesized summaries is map. Furthermore, the nested variants of CONSUL operators occur more commonly than their non-nested variants.

Finally, to answer our third research question, we give some examples of Solidity loops for which SOLIS was unable to generate a useful summary despite exploring the whole search space (i.e., no time-out). As expected, there are two reasons why SOLIS fails to summarize a loop’s behavior: (1) the semantics of the loop cannot be captured by the CONSUL DSL, or (2) synthesis fails due to the incomplete decomposition heuristics employed by SOLIS.

To understand the latter issue, consider the last example from Table 4. Here, because there is a dependency between the loop’s multiple side effects (e.g., the last fold operation depends on the earlier map operation), we cannot generate the summary in a compositional way, so the decomposition strategy described in Section 4.3 fails to work. As a result, SOLIS is unable to capture the loop’s effect on the variable called total even though the loop’s behavior can be precisely summarized in the CONSUL DSL.

Beyond limitations of the SOLIS tool chain, there are, of course, also Solidity loops that fundamentally cannot be expressed in CONSUL. We show three such examples in Figure 7.

For instance, consider the first loop from Figure 7. At first glance, the loop appears to do a fold over the array called _data. However, the expression on the right-hand side includes subtraction of the iterator i, which cannot be captured by any of the accumulator functions in CONSUL.

The second loop in Figure 7 does not access a collection as most loops do (see Section 2), but instead performs complicated arithmetic operations on the two integers first and last. However, we intentionally did not try to capture such numeric summaries in the CONSUL DSL because (a) such behavior is not as common, and (b) there is already a rich literature of program analysis techniques for inferring numeric invariants (e.g., based on abstract interpretation[10,36], constraint solving[9,23], Craig interpolation[24,35], abduction[11,33] etc.).

The third loop in Figure 7 involves a conditional whose else branch contains a nested loop. While the nested loop has a summary that is expressible in CONSUL (and which can also be synthesized by SOLIS), the behavior of the outer loop does not appear to be expressible in our summarization language.

**Result for RQ3**: There are two reasons why SOLIS may fail to generate a useful loop summary. One reason is due to the incomplete decomposition heuristic described in Section 4.3. Another reason is the presence of complex arithmetic or nested loops that result in behavior that is not easily expressible in the CONSUL DSL.

---

1Part of the reason that require is so common is due to the fact that SOLIS summarizes require statements with conjunctions in the Solidity code with multiple require statements.
5.3 Threats to validity
We believe there are two major threats to the validity of our conclusions, which we explain below.

Benchmark Selection. As mentioned earlier, we were able to evaluate SOLIS on 1220 of the 22,685 loops considered in Section 2 due to limitations of our implementation and time restrictions. However, we believe that our benchmarks are representative enough in terms of both complexity and diversity.

Validity of Summaries. As mentioned in Section 4, SOLIS uses bounded verification when checking equivalence. In theory, this means that some of the summaries synthesized by SOLIS may not actually be equivalent to the original loop. However, based on our manual inspection of randomly sampled summaries, we find that the loops synthesized by SOLIS are indeed equivalent to the original loop.

DSL Construction. Since the constructs of CONSUL in Figure 3 are designed manually by us, they may not be representative enough to cover all of the common loop patterns found in Solidity. To mitigate this concern, our DSL design was guided by static analysis, semantic clustering, and random sampling as detailed in Section 2.

6 RELATED WORK
Program analysis for smart contracts has been an active research topic in recent years. In what follows, we survey recent research on smart contract analysis as well as a few other topics relevant to this work.

Program analyzers for smart contracts. Existing tools for vetting smart contracts are based on a variety of different approaches, including symbolic execution [1, 2, 34, 38], abstract interpretation [20, 45], interactive theorem proving [21, 25], and testing [26]. To the best of our knowledge, none of these existing techniques aim to infer precise loop invariants that capture a loop’s side effects on contract state. Thus, existing techniques are either unsound or grossly imprecise in the presence of loops, or they require manually provided invariants from the user. We view this work as being complementary to existing efforts in the space of program analysis for smart contracts. In particular, our work brings clarity about the nature of loops that occur in smart contracts and takes a first step towards automatically inferring their semantics.

Loop summarization. Generally speaking, loop summarization refers to the task of replacing a loop \( L \) with a loop-free code snippet \( S \) (e.g., that over-approximates the actual behavior of the loop [19, 29, 42]). Loop summarization has been shown to be very helpful in program analysis, both in the context of verification [28] as well as symbolic execution [19]. Loop summarization has also been shown to be effective for program optimization. For example, the QBS tool by Cheung et al. improves program performance by summarizing Java loops using SQL statements [8]. This work is particularly related to ours in that they take a program synthesis based approach towards loop summarization; however, they perform synthesis using constraint solving as opposed to typed directed search. More generally, compared to all existing work on loop summarization, the key contribution of this paper is to provide a formalism (i.e., domain-specific language) that can be used to summarize behaviors of Solidity loops. In addition, we also take a first step towards loop summarization for smart contracts.

Equivalence checking. Checking equivalence between a pair of programs is a classical relational verification problem and arises in many different contexts, including translation validation [37], differential program analysis [31, 46], and regression verification [13]. Generally speaking, there are two different approaches to checking equivalence. One approach is to construct a so-called product program \( P \) such that two programs \( P_1 \) and \( P_2 \) are equivalent if and only if \( P \) does not have failing assertions [5, 6, 32]. An alternative approach is to utilize program logics, such as relational Hoare logic, that can be used to directly prove equivalence [7, 43]. In this work we take the former approach and construct a very simple product program through sequential composition. This simple approach is sufficient in our setting because our equivalence checker is based on bounded verification. While more sophisticated product construction mechanisms may better facilitate unbounded verification of Solidity programs, this is an orthogonal problem that we leave to future work.

Program synthesis. This work is related to a long list of research papers on program synthesis in that we automatically synthesize a DSL program that satisfies the given specification (in our case, the original Solidity loop). While program synthesis is a very active research area with a multitude of different techniques and applications [4, 14, 15, 17, 22, 44], our work is most similar to inductive synthesizers based on explicit search [4, 14, 15, 17]. More concretely, similar to prior efforts in this space [16, 17, 41], our synthesis technique uses type-based reasoning to significantly prune the search space but also utilizes basic static analysis of the original loop to further reduce the space of candidate summaries.

7 CONCLUSION AND FUTURE WORK
In this paper, we performed a large-scale investigation of loops found in Solidity smart contracts, including both their syntactic and semantic features. Based on these findings, we proposed a DSL for capturing common behaviors of Solidity loops and built a program synthesis toolchain called SOLIS for automatic loop summarization in smart contracts. Our experiments indicate that at least 56% of loops can be summarized using our proposed DSL and 81% of these summaries are equivalent to the original loop.

There are several directions for future work. First, while we have taken a first step towards loop summarization in smart contracts, our proposed tool chain sacrifices completeness for better scalability and therefore fails to generate summaries for larger loops even though an equivalent CONSUL summary exists. Thus, we plan to investigate better loop summarization techniques in future work. Second, we are interested in integrating our loop summarization method with existing verification tools to further automate the process of proving functional correctness of smart contracts.

REFERENCES

A SEMANTIC FEATURES USED FOR CLUSTERING

In this section, we describe the semantic features we used when performing our clustering analysis. These features are obtained by constructing a type of program dependence graph (PDG) and extracting various types of information from the paths of the corresponding PDG.

For our purposes, a program dependence graph \( G \) is a tuple \( (N, E_C, E_D) \) where \( N = N_I \cup N_W \cup N_R \) and \( N_I, N_W, \) and \( N_R \) correspond to variables of array indices, variables that are written to, and variables (other than those that are used as array indices) that are read from, respectively. \( E_C \) are edges from \( N_W \) to \( N_R \), and \( E_D \) are edges from nodes in \( N_I \) to \( N_W \). Intuitively, an edge \((u, v) \in E_D\) indicates that a variable \( u \) is used as an index into collection \( v \). An edge in \( E_C \) models other types of read-write dependency.

Figure 8 gives a few examples of Solidity loops and their corresponding PDG representation. Specifically, nodes in \( N_I, N_W, \) and \( N_R \) are colored in red, purple, and grey, respectively. Also, dashed arrows encode edges in \( E_D \) while solid arrows denote edges in \( E_C \).

Once we construct the graph representation, we extract the following features from the PDG:

Path related features. The following 8 features are related to paths in the PDG:

- Maximum and minimum number of any path from a a node in \( N_I \) to a node in \( N_W \).
- Maximum and minimum number of \( E_D \) path from a node in \( N_I \) to a node in \( N_W \).
- Maximum and minimum number of \( E_C \) paths from a node in \( N_I \) to a node in \( N_W \).
- Maximum and minimum number of mixed paths (involving both \( E_D \) and \( E_C \) edges) from a node in \( N_I \) to a node in \( N_W \).

Degree related features. The following 24 features are related to degrees of nodes in the PDG:

- Maximum and minimum number of mixed-edge in-degrees of a node in \( N_W \).
- Maximum and minimum number of mixed-edge out-degrees of a node in \( N_W \).
- Maximum and minimum number of \( E_D \)-only in-degree of a node in \( N_W \).
- Maximum and minimum number of \( E_D \)-only out-degree of a node in \( N_W \).
- Maximum and minimum number of \( E_C \)-only in-degree of a node in \( N_W \).
- Maximum and minimum number of \( E_C \)-only out-degree of a node in \( N_W \).
- Maximum and minimum number of mixed-edge in-degrees of a node in \( N_I \).
- Maximum and minimum number of mixed-edge out-degrees of a node in \( N_I \).
- Maximum and minimum number of \( E_D \)-only in-degree of a node in \( N_I \).
- Maximum and minimum number of \( E_D \)-only out-degree of a node in \( N_I \).
- Maximum and minimum number of \( E_C \)-only in-degree of a node in \( N_I \).
- Maximum and minimum number of \( E_C \)-only out-degree of a node in \( N_I \).

Other features. The following 5 features are related to degrees of nodes in the PDG:

- Maximum and minimum number of cycles involving a single \( N_W \) node in the PDG.
- Maximum, minimum and average path length from an \( N_I \) node to an \( N_W \) node in the PDG.

B SEMANTICS OF DSL CONSTRUCTS IN TERMS OF SOLIDITY CODE

In this Appendix, we provide a more formal semantics of CONSUL constructs in terms of their translation to Solidity code. In particular, the left column in Table 4 shows a CONSUL construct and the right column shows its corresponding Solidity equivalent.

- Maximum and minimum number of \( E_C \)-only out-degree of a node in \( N_I \).
Figure 8: Modeling the semantics of loops using program dependence graph.

<table>
<thead>
<tr>
<th>CONSUL Construct</th>
<th>Solidity Loop Equivalent</th>
</tr>
</thead>
</table>
| m4 = zipNestedASym(m1, m2, m3, φ, f) | for(k ∈ keys(m3) ∧ φ(k, m1[m3[k]])) {  
|                         |   m4[m3[k]] = f(m1[m3[k]], m2[k]); }                      |
| m4 = zipNestedSym(m1, m2, m3, φ, f) | for(k ∈ keys(m3) ∧ φ(k, m1[m3[k]])) {  
|                         |   m4[m3[k]] = f(m1[m3[k]], m2[m3[k]]); }                  |
| m3 = zip(m1, m2, φ, f) | for(k ∈ keys(m1) ∧ φ(k, m1[k])) {  
|                         |   m3[k] = f(m1[k], m2[k]); }                              |
| m3 = mapNested(m1, m2, φ, F) | for(k ∈ keys(m2) ∧ φ(k, m1[m2[k]])) {  
|                          |   m3[m2[k]] = F(m1[m2[k]]); }                            |
| m2 = map(m1, φ, F) | for(k ∈ keys(m1) ∧ φ(k, m1[k])) {  
|                          |   m2[k] = F(m1[k]); }                                    |
| v = foldl(m, φ, f, acc) | v = acc; for(k ∈ keys(m) ∧ φ(k, m[k])) {  
|                          |   v = f(v, m[k]); }                                     |
| v = foldlNested(m1, m2, φ, f, acc) | v = acc; for(k ∈ keys(m2) ∧ φ(k, m1[m2[k]])) {  
|                          |   v = f(v, m1[m2[k]]); }                                 |
| require(m, φ₁, φ₂) | for(k ∈ keys(m) ∧ φ₁(k, m[k])) {  
|                          |   require(φ₂(k, m[k])); }                                |
| requireNested(m1, m2, φ₁, φ₂) | for(k ∈ keys(m2) ∧ φ₁(k, m1[m2[k]])) {  
|                          |   require(φ₂(k, m1[m2[k]])); }                           |
| transfer(m1, m2, F, φ) | for(k ∈ keys(m1) ∧ φ(k, m1[k])) {  
|                          |   transfer(m1[k], F(m2[k])); }                           |

Table 7: CONSUL construct equivalents. We omit lambdas to improve readability.