Bytecode Skeletons for Sample Selection in the Analysis of Blockchain Programs

First Author

***Abstract*—To evaluate analysis tools for blockchain programs or to analyze an entire ecosystem of blockchain programs, representative samples are needed. Typically, samples are randomly selected from programs deployed during a particular period or published on web sites. Depending on the selection strategy, the quality of the analysis results may differ greatly.**

In this paper, we propose a selection method for smart contracts on Ethereum based on bytecode normalization. For each program, we compute a skeleton by removing parts with no or little effect on its functionality. Programs with the same skeleton are considered equivalent, and only one representative needs to be considered. We empirically evaluate its effect on the results of common bytecode analyzers. The proposed approach not only makes full coverage feasible, but also reduces sample size, redundancy, and bias. It sufficiently preserves the functionality and can even improve analysis results.

***Index Terms*—Bias, Code analysis, Code normalization, Ethereum, EVM, Smart contracts**

I. Introduction

Programs on a public blockchain, also known as smart contracts, execute in a hostile environment: the code is open to inspection, everyone can interact with it, updates are difficult, and the programs often handle valuable assets, which incentivizes attacks. Program authors, chain analysts, users and abuser are equally interested in analyzing the code (though with different objectives). Consequently, we find numerous tools aiming at security issues or the detection of other code properties.

A recurring task is to select representative data samples to evaluate, compare or train tools, or to gain insights into a blockchain ecosystem. Depending on the purpose, various sampling strategies are employed, like fixing a time window, picking randomly from large data bodies, or filtering with some code metrics. However, redundancies and potential biases are often not made explicit. Reasons may be that the focus is on the presentation of a method rather than on the robust evaluation of its implementation, or that the technological specifics of the application domain are not sufficiently familiar.

This neglect is a problem, since we observe a high degree of code duplication on blockchains like Ethereum. Code is redeployed in large numbers, in identical or slightly modified form. To make things worse, the multiplicity is not equally distributed, but follows a long tail distribution: most programs are unique, but a small number of programs occur in such a large number (of variants) that naive sampling strategies predominantly reflect these few and barely capture the diversity. This introduces bias into studies, but also consumes resources that could be invested in less redundant samples covering a broader spectrum.

While it is easy to eliminate exact copies, it is intrinsically difficult to determine whether two different codes are sufficiently similar to be considered the same with respect to a certain aspect. As an example, the difference between a program containing a vulnerability and its fixed counterpart is often so subtle that they seem to be functional twins, but they are not equivalent in the context of vulnerability detection. Thus, methods for redundancy elimination need to be conservative, equating programs only when there is a strong case for doing so. Moreover, the methods need to be evaluated in the context of the application domain, since programs may behave the same in one setting, but may need to be viewed as different in another.

Our Approach. In this work, we investigate bias in code from Ethereum's main chain, propose two methods for redundancy elimination, and evaluate them in the context of automated program analysis. The two reduction techniques consist in

- stripping parts that do not affect program execution, and
- normalizing parts that have only little impact on the functionality.

Research questions.

- Q1 How feasible is the approach? We argue the feasibility by the effort of detecting redundancy.
- Q2 How conservative is it? We empirically evaluate the divergence of analysis results when our approach is applied.
- Q3 What are the effects on the bias in sample selection? We demonstrate the bias reduction of our approach.

Empirical evaluation. Ethereum is the major platform for smart contracts in terms of the number of applications, tools, market cap, attacks, counter-measures, and academic studies. Therefore, we start our analysis from the almost 50 million deployments of smart contracts on Ethereum's main chain up to block 14 000 000 (January 2022). As application domain, we choose the automated analysis of blockchain programs. More specifically, we compare the behavior of 13 bytecode analyzers integrated in the framework SmartBugs[1] on programs identified as equivalent by our approach.

Benefits: Our approach of bytecode normalization

- allows for full coverage of entire blockchain ecosystems,
- reduces redundancy and bias, and
- enhances analysis results.

[1]https://github.com/smartbugs/smartbugs

II. Contract Creation and Bytecode Data

To deploy a contract on Ethereum, the Ethereum Virtual Machine (EVM) executes a create operation, which requires the *deployment code* of the contract. It consists of an active part, which typically initializes the environment for the new contract and returns the pointer to a memory area with the actual *runtime code*. The EVM then stores this runtime code at the address of the new contract. The deployment code may assemble the runtime code arbitrarily, but typically just copies code following its active part.

The majority of Ethereum contracts are written in Solidity, an object-oriented programming language. The so-called constructor and any global initializations compile to the active part of the deployment code, whereas all other parts of the source file compile to the runtime code, which is appended to the active part. Finally, the compiler appends *metadata*, which contains a hash identifying the original source code and version information, but has no influence on the behavior of the contract. Changing any character in the Solidity file, including comments and the newline encoding, alters the metadata and leads to superficially different deployment and runtime codes.

III. Related Work

In this section, we position our work within the field of bytecode similarity. Then we mention work on the similarity of EVM bytecode. Finally, we focus on closely related work regarding the normalization of bytecode.

Analysis of Bytecode Similarity. Haq et al. [1] recently surveyed over 20 years of research into the analysis of binary code similarity. We refer readers interested in bytecode similarity in general to this source. To position our work, we list Haq et al.'s classification of code similarity and highlight the properties characterizing our work.

- Type of comparison: identical (same syntax), **equivalent (same semantics)**, similar (similar syntax, structure, or semantics according to some measure)
- Granularity of code: instructions, basic blocks, functions, **program**
- Number of compared inputs: one-to-one, one-to-many, **many-to-many**
- Similarity types: syntactic (instructions), structural (graph of code), **semantic (functionality)**.
- Code normalization: **operand/instruction replacement or removal**
- Analysis type: **static (bytecode only)**, dynamic (execution of code, traces), hybrid analysis (both static and dynamic), dataflow analysis (ways of value propagation)

EVM Bytecode Similarity. For the purpose of finding semantically similar contracts, several approaches have emerged: we find graph-based approaches like [2]–[6] or approaches that utilize the interfaces exposed by smart contracts like [7]–[10]. As they do not address equivalence, but rather functional similarity, they are beyond the scope of this paper.

Bytecode Normalization. Normalization of bytecode is a widely used step in reducing duplicates. A straightforward possibility is to strip certain values (while potentially losing contextual information) as is done in e.g. [11], [12]. Koo et al. [13] suggest a method of normalization that balances the amount of tokens by applying rules that retain rich code information during stripping.

For EVM bytecode, normalization techniques are used in [14], [15] He et al. [14] remove the creation and Swarm code parts and "all assigned values in assignment statements and function calls". Di Angelo et al. [15] employ a similar approach: Solidity meta-data, constructor arguments and `PUSH` arguments are replaced by zeros, then trailing zeros are stripped. Both works provide verbal justifications for their techniques only.

IV. Approach

In this section we present two techniques for normalizing code: the removal of metadata and the normalization of `PUSH` operations. The aim is to detect functionally equivalent code by computing normal forms and testing the latter for identity.

A. Removing Metadata

Early on, the Solidity compiler started to append metadata to the runtime bytecode, in the form of a dictionary with one to three entries, encoded in Concise Binary Object Representation (CBOR). The entries may include a hash identifying the source code on a decentralized storage like IPFS, or the version of the Solidity compiler. Any reformatting of the source code or the use of another compiler version will change the metadata, even if the actual code stays the same. As the metadata gets neither executed nor read (except during deployment), contracts differing just in their metadata are functionally equivalent.

To identify such equivalences, we may either replace the metadata by some fixed values, like by a stretch of zeros of the same length, or remove the metadata. The first alternative leaves the code executable, as an address referencing a position after the metadata stays correct. However, if the metadata sections are of different lengths, replacing them by zeros leaves the codes different. Removing the metadata improves the situation if the metadata occurs at the end, but still suffers from the problem that after metadata of different lengths, the address of logically equivalent positions differ, so code referring to it will differ, too. This issue will be addressed below by the second type of normalization.

Algorithm 1 detects the metadata. It scans the code for one of the bytes `0xA1`, `0xA2` or `0xA3` (lines 4–5), as they indicate the start of a CBOR-encoded mapping with one to three items. This check speeds up detection, as the bytes are rare and the actual decoding is tried in a few places only, but the check also ensures that the structure is a mapping if decoding is successful. Lines 6–10 perform the actual decoding. Moreover, they interpret the two bytes following the CBOR structure as a number, as the Solidity compiler appends the length of the structure. If the decoding or the access to the length fails, an

Algorithm 1 Extracting the metadata: METADATA($code$) returns a list of triples (m, s, e) such that m is the metadata encoded by $code[s : e]$. DECODECBOR decodes the initial segment of its argument and returns the decoded structure as well as the number of used bytes.

```
function METADATA($code$)
    $i$, $metadata$ := 0, [ ]
    while $i <$ length($code$) do
        if $code[i] <$ 0xA1 or $code[i] >$ 0xA3 then
            $i := i + 1$; continue
        try    ▷ Does $code[i]$ start a dict. with 1–3 entries?
            $(m, l_1) :=$ DECODECBOR($code[i : ]$)
            $l_2 := code[i+l_1] * 256 + code[i+l_1+1]$
        catch
            $i := i + 1$; continue
        if $l_1 = l_2$ and one of the keys bzzr0, bzzr1,
          ipfs or solc occurs in dictionary $m$ then
            $metadata$.append( $(m, i, i+l_1+2)$ )
            $i := i + l_1 + 2$
        else
            $i := i + 1$
    return $metadata$
```

exception occurs and scanning continues at the next position. Otherwise, the checks in line 11 ensure that we have indeed found Solidity metadata, by verifying that the length of the structure equals the number following it, and that one of a few characteristic keys occurs in the mapping.

B. Normalizing PUSH Operations

Algorithm 2 NORMALIZEPUSH($code$) replaces PUSH operations and their argument by the opcode PUSH1. LENGTHARG returns the number of argument bytes, i.e., 1–32 for PUSH1–PUSH32, and 0 otherwise.

```
function NORMALIZEPUSH($code$)
    $i$, $opcodes$ := 0, [ ]
    while $i <$ length($code$) do
        if LENGTHARG($code[i]$) $> 0$ then
            $opcodes$.append(0x60)        ▷ 0x60 = PUSH1
        else
            $opcodes$.append($code[i]$)
        $i := i +$ LENGTHARGS($code[i]$) $+ 1$
    return joined($opcodes$)
```

EVM bytecode consists of one-byte-instructions that obtain their arguments from one of the memory areas, most prominently the stack. There is one notable exception: The operations PUSH1 to PUSH32 are followed by a literal of 1 to 32 bytes that represents the value to be pushed on the stack. These operations with their data are often responsible for variations in otherwise identical code. As an example, hard-coded addresses typically appear as the argument of a PUSH20 operation, but in the case of leading zeros, it may also be a PUSH15–PUSH19 instruction. Likewise, many token contracts are identical except for a few constants that appear as PUSH arguments. Such local variations have global effects, as the position of jump targets depends on the number of bytes preceding it. However, the address of such targets appears, more often than not, as the argument of another PUSH operation that precedes a JUMP. By replacing all PUSH operations and their arguments with a single fixed byte – we choose PUSH1 – we get rid of such variations (algorithm 2).

Algorithm 3 Computing the skeletons: SKELETONS($code$) returns a pair $(preskel, skel)$, where $preskel$ is the $code$ without metadata, and $skel$ additionally normalizes the PUSH operations.

```
function SKELETONS($code$)
    $i$, $preskel$, $skel$ := 0, [ ], [ ]
    for $(\_, s, e)$ in METADATA($code$) do
        $preskel$.append($code[i : s]$)
        $skel$.append(NORMALIZEPUSH($code[i : s]$))
        $i := e$
    $preskel$.append($code[i : ]$)
    $skel$.append(NORMALIZEPUSH($code[i : ]$))
    return (joined($preskel$), joined($skel$))
```

C. Skeletons and Families

For the discussion and evaluation of our approach, we use the following terms.

Skeletons and Pre-Skeletons: Given a bytecode, we compute its *skeleton* by removing metadata and by normalizing PUSH operations. To analyze the effect of this reduction, we consider also the *pre-skeleton*, which is the intermediate result after removal of the metadata but before PUSH-normalization.

Algorithm 3 computes the pre-skeleton and the skeleton. In line 3, we call METADATA to obtain the start, s, and the end, e, of each metadata section. The loop (lines 3–6) collects the code between metadata sections in $preskel$, and the code with normalized PUSHes in $skel$. When the loop terminates, we add the code snippet after the last metadata. Finally, line 9 returns the pre-skeleton and skeleton as byte strings, which are a concatenation of the respective list elements.

Families and Pre-Families: The *family* of a skeleton is the collection of all bytecodes that normalize to this skeleton. The *size* of a family is the number of bytecodes in the collection. The *creation block* of a family is the earliest block, where some member of the family has been deployed. Likewise, the *pre-family* of a pre-skeleton is the collection of all bytecodes with this pre-skeleton.

Observe that there are three types of many-one relations between deployments, bytecodes, pre-skeletons, and skeletons. Each bytecode may get deployed multiple times in identical form, each pre-skeleton may correspond to several bytecodes differing in their metadata, and each skeleton may be obtained from several pre-skeletons by PUSH-normalization. Thus, each family can be partitioned into one or more disjoint pre-families.

D. *Feasibility of the Approach (Q1)*

The function METADATA (Algorithm 1) consists of two nested loops: The outer loop iterates over the code, while the inner loop, hidden in DECODECBOR, iterates over code sections starting with `0xA1`, `0xA2` or `0xA3`. In the computationally best case, each of these three bytes starts a valid metadata section, so the code is scanned just once by alternating between the outer while-loop and the decoding loop. In the worst case, the code is full of the three bytes, each starting a large CBOR-structure that fails the check in line 11, such that the structure is abandoned and the counter i is incremented by one. Hence, the time complexity of METADATA is $O(\text{length}(code)^2)$.

In practice, the trigger bytes `0xA1`, `0xA2` and `0xA3` occur infrequently, either as instructions `LOG1`, `LOG2` and `LOG3` that are generated from Solidity's emit statements, or as data. For the 514 893 distinct bytecodes deployed until block 14 M with an average length of 5 772 bytes, the trigger bytes occur 10.8 times per contract on average, with 0.92 starting a metadata section. Thus, we encounter roughly one metadata section and ten failed decoding attempts per contract, which makes the algorithm linear rather than quadratic on real data.

`PUSH` normalization is linear, as algorithm 2 scans the code exactly once while copying or skipping bytes. Skeleton and pre-skeleton are computed by executing both algorithms in succession (algorithm 3), hence the skeletons can be computed in linear time plus an overhead for a few failed decoding attempts. On a single processor at 2.2 GHz, a Python implementation of the algorithms computes the skeletons for 514 893 bytecodes in 2 065 seconds, amounting to an average of 4 ms per contract.

V. REDUCTION THROUGH NORMALIZATION

In this section, we investigate the distribution of bytecodes and the effects of our approach.

A. *The Data*

By running an OpenEthereum[2] node, we collected the runtime codes of all contracts that were successfully deployed on Ethereum's main chain up to block 14 M (13 Jan 2022).

TABLE I: Deployments up to Block 14 M

Metric	# Contracts
Deployments	48 262 411
Distinct deployment codes	2 206 793
Distinct runtime codes	514 893

Table I gives an overview of the deployment activities. The 48.3 M contract creations involved 2.2 M different deployment codes, generating a total of 0.5 M distinct runtime codes. 99.0 % of these codes originate from the Solidity compiler (as determined by characteristic byte sequences), with the source code for 46.5 % being actually available on Etherscan[3].

[2]https://github.com/openethereum/openethereum
[3]https://etherscan.io

Visualization: For visualizing the data on a timeline, we use the creation block as a proxy for time. For code representing a collection of similar ones, we choose the earliest block (i.e., the smallest block number) where any of the codes has been created. As an example, runtime code that has been deployed multiple times is associated with the earliest deployment. Likewise, a bytecode representing a family of codes is located at the earliest deployment block of any code in the family.

We aggregate the data in bins of 100 000 blocks, resulting in 140 points on the horizontal axis. Per bin, we plot the number of bytecodes in absolute or relative terms. A bin roughly corresponds to two weeks of blockchain activity. When interpreting the figures from the point of data sampling with a time window, it is convenient to think of a few neighboring bins as the sample (if all the data is picked) or as the data being sampled.

B. *Reduction Achieved*

Applying our approach of code normalization, we achieve a substantial reduction in the number of bytecodes (table II). We see a reduction by 30 % when ignoring the metadata, i.e., considering pre-families instead of distinct runtime bytecodes, and a reduction by 53 % when additionally normalizing `PUSH` operations, i.e., considering code families instead of distinct runtime bytecodes.

TABLE II: Reduction of Bytecodes via Normalization

Metric	# Contracts
Distinct runtime bytecodes	514 893
Pre-families	364 099
Code families	241 293

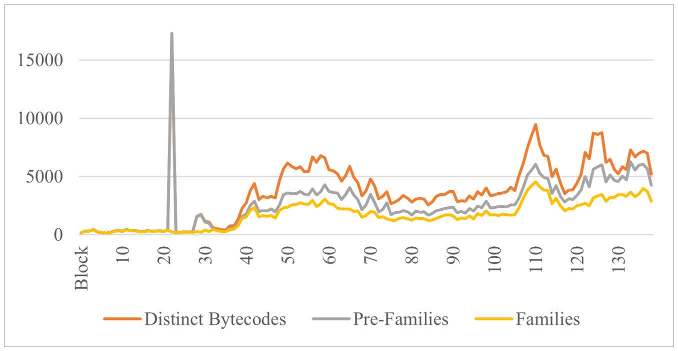

Fig. 1: Reduction of bytecodes via normalization.

Fig. 1 depicts the runtime codes, the pre-families and families over time. We see that the plots flatten when going from runtime codes to families, indicating that the rate of novel contracts changes less than the numbers of runtime codes would suggest.

Another effect of forming families is that singular events disappear. At bin 23, we see a spike of more than 15 000 bytecodes (with the gray line for pre-families covering the orange one). A DoS attack targeting underpriced instructions involved a series of almost 16 000 contracts differing in a single Ethereum address, hardcoded with a `PUSH20` instruction. In the family plot (yellow), this difference factors out, such that the contracts count as just a single one.

C. Family Sizes

The 241 293 families come in different sizes. Fig. 2 shows the distribution of 200 341 singletons, i.e., families with just one member, and of 40 952 families with more than one. The latter, though just a sixth of the families, comprise 314 552 distinct bytecodes in total, with 16 374 in the largest family.

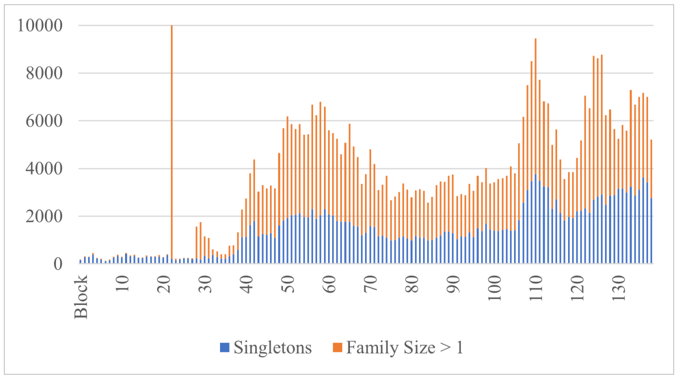

Fig. 2: Distinct bytecodes without another family member (singletons, blue) and with at least one sibling (orange) as a stack plot. The height of the stack corresponds to the number of distinct bytecodes per bin.

Table III gives an overview of the largest families and their purpose. We discuss a few to understand the reasons for their proliferation.

TABLE III: The largest families, with the topmost 11 in size and the topmost 8 in number of deployments (with one overlap)

Family Size	# Deployments	Purpose of Contract
16 374	2 524 507	proxies, mostly used for wallets
15 992	15 992	DoS attack at blocks 2 375 513 ff.
5 575	5 575	UniswapV3Pool
4 080	5 144	call forwarder, with 1 wei to creator
3 681	3 817	ERC20 token
3 272	3 273	write to memory (no state change)
3 257	3 257	library reverting unconditionally
2 114	2 626	ERC20 token
2 000	3 289	crowdsale pricing
1 937	1 937	Ether timelock
1 782	2 239	token timelock
...	...	...
907	1 098 096	contract reverting unconditionally
...	...	...
159	21 732 218	gas tokens
...	...	...
10	1 202 089	Ambi wallet
...	...	...
7	1 151 975	BitGo forwarder
...	...	...
5	1 657 760	Bittrex controlled wallet
...	...	...
2	1 515 594	BitGo forwarder
...	...	...
1	4 369 789	self-destructing creation code

The largest family with 16 374 codes, and the third largest in terms of deployments, is a type of proxy contract. Proxies delegate incoming calls to a main contract that implements the functionality once for all. The proxies here contain the target address hard-coded, which makes the 16 k contracts differ in a single `PUSH` and thus members of the same family. The much larger number of deployments indicates that many proxies also share the target address.

The members of the second largest family cause the spike in fig. 1; see the discussion of the figure in the last section for details.

From the third line onward, we find more conventional contracts, like UniswapV3Pool, ERC20 tokens and wallets. They differ in some addresses and constants, e.g. when interacting with specific tokens or external libraries, whose addresses need to be known. Observe that the differences are more subtle than just a different initialization, as parameters like token name, token supply and fixed administrators are set during deployment, which is factored out by considering the runtime bytecode.

Table III illustrates that analyzing families instead of codes or deployments helps reduce mass phenomena and focus on more diverse activities. Gas tokens are tiny programs that compare the caller address to an hard-coded one, and if they match, self-destruct, which until recently reduced the transaction costs for the caller. The almost 22 M deployments boil down to just 159 codes, all belonging to the same family. The second largest family with 4.4 M deployments are contract creations that self-destruct at the end of the deployment phase instead of returning runtime code, resulting in an empty code and thus forming a single family. These contracts are scripts to be executed just once, usually to outsmart a contract that relies on block variables or `EXTCODESIZE`. If a script does not clean up by self-destruction but terminates regularly, the Solidity compiler generates a runtime code that reverts for every call. We find 1.1 M of such contracts with 907 different codes, all belonging to the same family. Likewise, we find 3 257 libraries without functions that revert for every call. As every Solidity library starts by `PUSH`ing its address, we have as many codes as deployments, but forming a single family.

VI. Empirical Evaluation

In this section, we empirically evaluate our approach by analyzing its effect on contract analysis tools. After presenting the study design, we compare the results of the analyzers regarding divergences in code families. Finally, we answer research question Q2.

A. Study Design

Claim. Based on our analysis in the previous section, we propose to select contract samples for experimental evaluations either from a reduced dataset, or to reduce the contracts after their selection by computing the skeleton of their bytecode and retaining only one contract per skeleton. We claim that for many applications, the members of a family will behave identical, as they only differ in irrelevant details. Therefore, our approach reduces the bias and the processing overhead incurred by duplicate data entries, without reducing data variety.

Exemplary Use Case. To support this claim, we consider the automated analysis of bytecode. We run several analyzers, mostly weakness detection tools, on a selection of bytecodes and compare the results for codes of the same family. If our hypothesis is correct, we should see only minor variations.

Selection of the Bytecode Sample. To obtain a bytecode sample with several members per family, we partition the bytecodes deployed on Ethereum's main chain up to block 14 M into families and pick one code per family as its representative. From the non-representatives, we select 2000 codes at random, which happen to belong to 1107 families. We add the representatives of these families and end up with a sample of 3107 bytecodes. By construction, our sample contains a minimum of two members per family. 903 families occur with exactly two members, while the largest group consists of 120 codes. The differences in size reflect the underlying distribution described in sect. V-C, with the 120 codes belonging to the largest family in table III.

Execution Environment. We employ the framework SmartBugs, which offers 13 tools for bytecode analysis. For each run of a tool on a bytecode, SmartBugs reports findings, errors, failures,[4] informational messages, and the computation time in a unified format. We execute the tools on an AMD Ryzen 3990X processor with 64 cores@2.2 GHz and 128 GB of main memory. We limit the execution time of a single run to 30 minutes and allot 1.5 CPUs and 20 GB of memory. For reasons explained below, we repeat the computations with modified bytecodes, where we replace the metadata by a stretch of zeros of equal length. In total, the 80792 runs take 216 CPU days.

B. Analysis of Results

We compare the results for members of the same family in two ways. On the one hand, we consider findings only, as they are the aim of the analysis. On the other hand, we compare all fields, including also errors, failures, messages, and run times.[5] If any two runs of a family differ regarding findings or in any of the fields, we mark the family as divergent.

Expected Results. For several reasons, we expect a small fraction of divergent families. First, the execution environment is not deterministic. As an example, the timeout refers to the wall time, not the actual runtime. Differences in scheduling may allow a tool to finish the analysis of a particular code in time, but may lead to a timeout for an equivalent code of the same family because of execution delays. Second, some analyzers use randomness or internal timeouts, which contributes to indeterminism. Third, metadata are indeed functionally irrelevant, while the arguments of `PUSH` operations are not entirely so. Components like constraint solvers may yield different results for different values.

[4]Errors are conditions detected by a tool, failures are uncaught exceptions.

[5]We consider the run times of a family as divergent, if the fastest and the slowest run are more than 5 minutes (20% of 30 minutes) apart.

Actual Results. We start by evaluating the effects of metadata and indeterminism in isolation. To this aim, we group the results by pre-families, considering only those with at least two members. As families typically split into several pre-families, many of which are singletons, we end up with 689 pre-families of size 2 or more. By definition, the codes of a pre-family are identical except for the metadata. If we replace the metadata by zeros, then the codes become identical, hence the results of a pre-family with zeroed metadata correspond to re-runs with the same input.

TABLE IV: The effect of metadata on analysis results. The numbers give the percentage of 689 pre-families with divergent results.

Preprocessing:	none		metadata zeroed	
Divergence in:	findings	all fields	findings	all fields
Conkas	1.2	1.6	0.7	0.9
Ethainter	0.0	0.1	0.0	0.0
Madmax	0.0	0.0	0.0	0.0
Mythril	0.6	0.7	0.1	0.1
Securify	0.1	2.2	0.4	0.5
Teether	0.0	1.0	0.0	0.0
Ethor	1.5	6.4	0.1	0.7
Honeybadger	0.0	19.9	0.1	0.1
Osiris	0.7	20.2	0.1	0.1
Oyente	0.3	20.2	0.3	0.2
Pakala	0.0	18.0	0.0	0.0
Maian	16.3	16.3	0.1	0.1
Vandal	6.2	6.8	0.1	0.1

Table IV lists for each tool the percentage of pre-families with divergent results. The columns to the left list the divergences for codes differing in their metadata, while the columns to the right correspond to runs with identical code and thus show the extent, to which indeterminism affects the results. The topmost group of tools behaves as expected, with the divergence roughly the same in both cases. The second group of tools shows low divergence for the findings, but high divergence regarding the other fields. The explanation is that during preprocessing, these tools scan the entire contract with its metadata, and issue a warning if they encounter a byte that does not correspond to a known opcode (even if the byte occurs in metadata). The subsequent static analysis, however, ignores the metadata, so the findings remain unaffected.

Pakala resembles the second group, but when keeping the metadata, we see a considerable number of pre-families, where some members finish just before the external timeout hits, while others take a few seconds longer and fail with a timeout.

Finally, the two tools at the bottom show divergent behavior even for the findings. Maian scans the code (and data) for specific instructions, and upon detecting them, performs a sometimes costly reachability analysis. So Maian's inability to skip metadata leads to divergent behavior in otherwise identical bytecode. For Vandal, the situation is similar, except that the tool constructs a control flow graph for each part of the bytecode, including data sections. As this extra work is time consuming, we see differences in both, findings and all fields.

TABLE V: The effect of PUSH operands and metadata on analysis results. The numbers give the percentage of 1107 families with divergent results.

Preprocessing:	none		metadata zeroed	
Divergence in:	findings	all fields	findings	all fields
Conkas	1.4	2.2	1.2	1.7
Ethainter	0.0	0.1	0.0	0.2
Madmax	0.0	0.0	0.0	0.0
Mythril	0.8	1.1	0.5	1.3
Securify	0.3	1.6	0.5	2.5
Teether	0.0	1.4	0.0	2.2
Ethor	3.1	8.3	1.5	1.9
Honeybadger	0.1	18.8	0.2	0.2
Osiris	1.0	19.4	0.3	0.4
Oyente	0.4	19.2	0.4	0.5
Pakala	0.0	14.9	0.0	13.6
Maian	13.6	13.6	0.1	0.1
Vandal	5.2	5.6	0.1	0.5

Table V shows the results for families, both with unmodified code and with metadata replaced by zeros. The situation resembles the one for pre-families. Zeroing the metadata removes a distraction that causes spurious behavior with some tools. The divergence of findings remains low in both cases.

C. Assessment of the Degree of Conservation (Q2)

Our evaluation confirms that the bytecode analyzers yield identical results for the members of a family, as demonstrated by the low values for the percentage of families with divergent findings (fourth column in table V). Surprisingly, the divergence that we observe (blue fields in columns 2 and 3) stems from differences in the metadata, which by design is irrelevant for the execution of bytecode, rather than from PUSH normalization. This phenomenon occurs with tools that apply some preprocessing to the entire byte string that does not distinguish between reachable and unreachable positions. A cheap solution to this issue would be to identify metadata with algorithm 1 and to replace it by zeros before handing the bytecode over to such tools.

VII. Bias in Sample Selection

We begin with an overview of sources of bias before arguing that random sampling of deployments or runtime code can result in a biased sample in terms of code diversity and bytecode age.

Sources of bias. When sampling contracts, the following factors may introduce a bias.

- Restricting the sampling window to a specific *block range* misses phenomena outside. In particular, the sample does not cover instruction sequences characteristic of compiler versions in use before or after the window, or EVM opcodes introduced at a later fork.
- *Mass phenomena*, i.e., large numbers of similar contracts, often deployed automatically, may lead to an overrepresentation of this type of contracts.
- The collection of *source codes on Etherscan* is biased in the sense that they were deliberately uploaded by the respective authors. This favors application types requiring users to trust it. Contracts for private use or automatically deployed contracts are underrepresented.

A. Bias Resulting from Family Size

Sampling the deployments: Suppose we want to sample the contracts on the Ethereum mainchain directly. As sampling window, we use one of the 100 000 block ranges that corresponds to a bin in our figures.

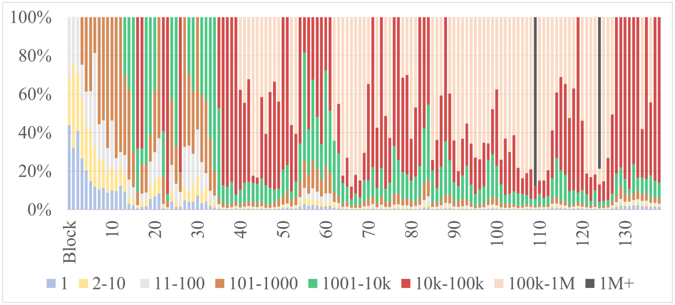

Fig. 3: Family size of all deployments.

Fig. 3 visualizes the distribution of family sizes per bin. Each stack corresponds to the contracts deployed in that block range. For each contract in a bin, we count the number of contracts possessing a runtime code from the same family, and color the contract according to this multiplicity using a logarithmic scale. As an example, the first bin in fig. 3 indicates that 44 % of the deployed contracts are singletons (blue), i.e., their bytecode has a unique skeleton within the bin, whereas 25 % deployments are contracts occurring in groups of 2–10 members of the same family (yellow), and the remaining 31 % contracts can be grouped into 11–100 family members (gray).

If we were to take a sample from a bin entirely blue, the selection would consist of bytecodes with pairwise distinct skeletons. For every other color, we may have duplicates, as there is more than one member of the family in the bin, but the selection would still be somewhat balanced, as all contracts occur with roughly the same multiplicity.

The actual distribution of family sizes in fig. 3, however, is a mixture of multiple ranges. Any random sample will reflect this imbalance and will be skewed.

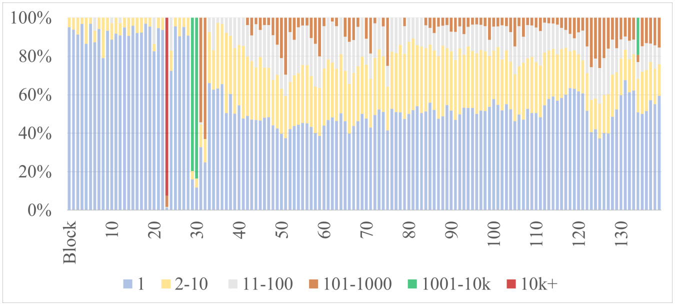

Fig. 4: Family size of bytecodes.

Sampling the runtime codes: The situation improves when considering only one deployment per runtime code. Fig. 4 shows the percentage of family sizes after this reduction. Sampling a collection of distinct runtime codes results in less redundancy than picking from all deployments, but still suffers from the disparity in family sizes that inevitably favors codes of larger families.

B. Bias Resulting from Code Age

In this section, we investigate the distribution of code age per bin. For a bytecode, we define its age as the difference (in blocks) between the earliest deployment of the bytecode (which defines its position on the timeline) and the first deployment of any code in its family. Certain code properties of a family, like the EVM instructions used or the code patterns generated by the compiler, are tied to this first deployment. Even if a bytecode of the family is deployed later on, its instruction set and code structures will reflect the earlier state of affairs.

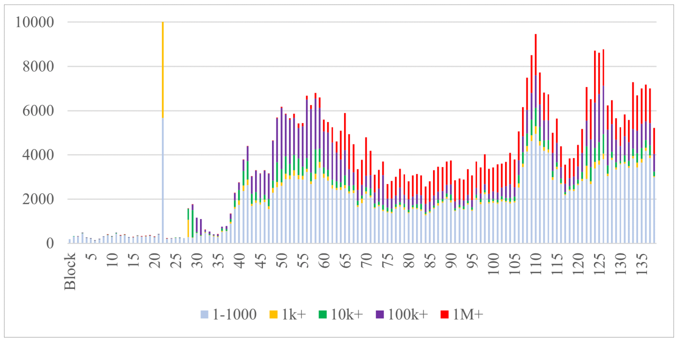

Fig. 5: Relative bytecode age in blocks. Each bin shows the number of distinct bytecodes. Age groups are stacked from younger to older. The peak near block 2.4 M has been clipped.

Fig. 5 depicts the age of bytecodes. The lowest portion of each bin shows the number of bytecodes with an age of up to 1 000 blocks, The other age groups are stacked on top, with the oldest ones being bytecodes with an age between 100 000 blocks and one million blocks (two weeks up to half a year, purple), and with an age of more than a million blocks (half a year or more, red).

Often it is preferable to sample deployments with their source code available on the platform etherscan.io, as Solidity source code is easier to analyze than EVM bytecode. Therefore we repeat the age analysis for 208 864 contracts with source code, where the age is computed from the deployment block of the specific contract and the earliest deployment resulting in the same runtime code.

Fig. 6 resembles fig. 5. In both, the majority of codes is younger than 100 000 blocks (lower three sections in each bin; light blue, yellow, and green), amounting to two weeks on the mainchain. However, virtually every bin also contains a non-negligible fraction of older contracts.

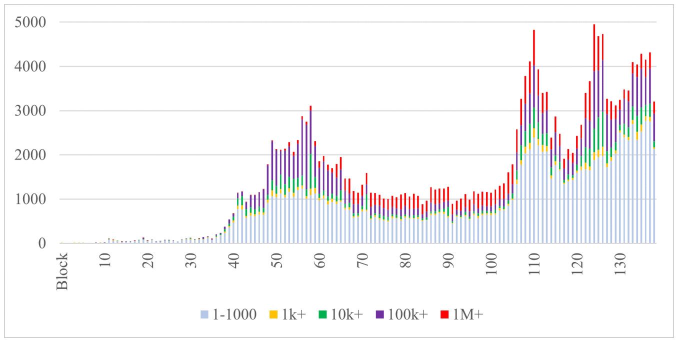

Fig. 6: Relative source code age in blocks. Each bin shows the number of the corresponding bytecodes. Age groups are stacked from younger to older.

C. Effects of the Proposed Approach (Q3)

Our method of using skeletons to group deployments and bytecodes into families allows us to detect bias resulting from family size or code age. At the same time, it also offers a method to counter them: Keep only the oldest contract per family in the sample. Moreover, the elimination of duplicates allows to cover larger block ranges without increasing the sample size. In the limit, a sample of 241 293 contracts captures all deployments up to block 14 M.

A bias not addressed by our approach is the one related to the motivation for providing source code on Etherscan.

VIII. Conclusion

In this work, we propose two techniques of bytecode normalization, the removal of metadata and the normalization of `PUSH` operations, for identifying similar contracts via their skeletons. The proposed approach is efficient as the skeletons can be computed in almost linear time. Compared to distinct bytecodes, the size of samples reduces by more than a half. Moreover, our approach is sufficiently conservative to yield only small differences in the analysis results for members of the same family. In addition, the concept of code families helps in identifying and reducing bias in data sets, both for bytecodes and source code.

Future work. Some versions of the Solidity compiler introduce minor changes in the bytecode without altering the functionality. These could be captured as well to further reduce bias and redundancy. A similar approach could be applied on source code level; this would allow us to reduce the redundancies in the contract data provided by Etherscan. Finally, for properties related to the deployment code (the Solidity constructor), we can extend the concept of code families to families of deployment code.

References

[1] I. U. Haq and J. Caballero, “A survey of binary code similarity,” *ACM Comput. Surv.*, vol. 54, no. 3, April 2021.

[2] I. Keivanloo, C. K. Roy, and J. Rilling, “Sebyte: Scalable clone and similarity search for bytecode,” *Science of Computer Programming*, vol. 95, pp. 426–444, 2014, special Issue on Software Clones (IWSC’12).

[3] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, “Eclone: Detect semantic clones in ethereum via symbolic transaction sketch,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 900–903.

[4] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, “Hunting vulnerable smart contracts via graph embedding based bytecode matching,” *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[5] D. Zhu, J. Pang, X. Zhou, and W. Han, “Similarity measure for smart contract bytecode based on cfg feature extraction,” in *2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI)*, 2021, pp. 558–562.

[6] D. Zhu, F. Yue, J. Pang, X. Zhou, W. Han, and F. Liu, “Bytecode similarity detection of smart contract across optimization options and compiler versions based on triplet network,” *Electronics*, vol. 11, no. 4, p. 597, 2022.

[7] M. Fröwis, A. Fuchs, and R. Böhme, “Detecting token systems on ethereum,” in *Financial Cryptography and Data Security*, I. Goldberg and T. Moore, Eds. Cham: Springer International Publishing, 2019, pp. 93–112.

[8] M. di Angelo and G. Salzer, “Assessing the similarity of smart contracts by clustering their interfaces,” in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020, pp. 1910–1919.

[9] M. di Angelo and G. Salzer, “Wallet contracts on Ethereum – identification, types, usage, and profiles,” *arXiv:2001.06909v2*, 2021.

[10] J. He, S. Li, X. Wang, S.-C. Cheung, G. Zhao, and J. Yang, “Neural-febi: Accurate function identification in ethereum virtual machine bytecode,” *Journal of Systems and Software*, vol. 199, p. 111627, 2023.

[11] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, “Safe: Self-attentive function embeddings for binary similarity,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 309–329.

[12] S. H. H. Ding, B. C. M. Fung, and P. Charland, “Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[13] H. Koo, S. Park, D. Choi, and T. Kim, “Binary code representation with well-balanced instruction normalization,” *IEEE Access*, vol. 11, pp. 29 183–29 198, 2023.

[14] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, “Characterizing code clones in the Ethereum smart contract ecosystem,” in *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 2020, pp. 654–675.

[15] M. di Angelo and G. Salzer, “Characterizing types of smart contracts in the Ethereum landscape,” in *Financial Cryptography and Data Security. FC’20, 4th Workshop Trusted Smart Contracts (WTSC)*, M. Bernhard, A. Bracciali, L. J. Camp, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, Eds. Springer, 2020, pp. 389–404.