Consolidation of Ground Truth Sets for Weakness Detection in Smart Contracts

Monika di Angelo(✉) and Gernot Salzer

TU Wien, Vienna, Austria
{monika.di.angelo,gernot.salzer}@tuwien.ac.at

Abstract. Smart contracts are small programs on the blockchain that often handle valuable assets. Vulnerabilities in smart contracts can be costly, as time has shown over and over again. Countermeasures are high in demand and include best practice recommendations as well as tools supporting development, program verification, and post-deployment analysis. Many tools focus on detecting the absence or presence of a subset of the known vulnerabilities, delivering results of varying quality. Most comparative tool evaluations resort to selecting a handful of tools and testing them against each other. In the best case, the evaluation is based on a smallish ground truth. For Ethereum, there are commendable efforts by several author groups to manually classify contracts. However, a comprehensive ground truth is still lacking.

In this work, we construct a ground truth based on publicly available benchmark sets for Ethereum smart contracts with manually checked ground truth data. We develop a method to unify these sets. Additionally, we devise strategies for matching entries that pertain to the same contract, such that we can determine overlaps and disagreements between the sets and consolidate the disagreements. Finally, we assess the quality of the included ground truth sets. Our work reduces inconsistencies, redundancies, and incompleteness while increasing the number of data points and their heterogeneity.

Keywords: analysis · benchmark · Ethereum · security · vulnerability

1 Introduction

To support the development of Ethereum smart contracts (SCs) and to analyze SCs that have been deployed, over 140 tools were released until mid-2021 [11], and new tools keep appearing. The sheer number of tools makes it difficult to choose an appropriate one for a particular use case. Moreover, it is difficult to assess the effectiveness of the many methods proposed, and to judge the relevance of various extensions. Tool comparisons can facilitate the selection process. However, many tool surveys are based on academic publications that focus on the methods employed by the tools, or on whitepapers of the tools themselves. For a thorough quality assessment of the tools, it is necessary to also install and systematically test the tools – preferably with an appropriate ground truth set of SCs.

© International Financial Cryptography Association 2024

Given the scarce availability of an appropriate ground truth, tool developers adopted the practice of comparing their tool to previous ones, often with the somewhat biased intention of demonstrating the superiority of their tool in a particular respect. This approach is justified by the need for an evaluation despite the lack of an established ground truth. However, there are major concerns about the validity of such evaluations.

Undetermined Quality of Tools: Since the point of reference is unclear, a comparison to something of unknown quality only provides relative information.

Dependence Between Tools: When a new tool builds on tools published earlier, there is a tendency to compare it to exactly those tools in order to show the improvements. With the quality of the base tool(s) not clearly determined, the relative quality assessment remains vague.

Ground Truth (GT). In our context, a *ground truth* for a particular program property is a set of smart contracts (given as source or bytecode) together with assessments that state for each contract whether it satisfies the property or not. As the term truth suggests, these assessments are supposed to be definitive and reliable. To foster trust into the ground truth, it may be accompanied by a specification of the process how the assessments were obtained (e.g. by expert evaluation) or by objective arguments for the assessments (e.g. by specifying program inputs that solicit behavior satisfying the property, or by showing that such inputs do not exist).

Goals and Approach. The primary goal of this work is to compile a unified and consolidated ground truth of SCs with manually labeled properties, starting from GT sets that are publicly available and documented. Ultimately, we aim at a uniformly structured collection of contracts with verified properties that harnesses the individual efforts that have been invested into the original datasets.

Unification. From related work, we collect benchmarks containing GT data. We extract information on the corresponding contracts (like address, source code, bytecode, location of the issue) as well as classifications (properties tested, assessments) and introduce a unique reference for every entry in the original dataset. We clean the data by repairing obvious mishaps and complete it using our database of source codes and chain data.

Consolidation. To consolidate the datasets, we introduce four attributes per contract: the address (with chain and creation block) if the contract has been deployed, as well as unique fingerprints of the source code, the deployment and the deployed bytecode. Based on these attributes, we determine and eliminate discrepancies within the individual datasets. Then, we map the classifications to a common frame of reference, the SWC[1] classes and the DASP[2] scheme. Relying again on the attributes, we determine overlaps between datasets, detect disagreements, and examine their cause.

Quality Assessment. Based on the taxonomy by Bosu et al. [2] for assessing data quality in software engineering, we assess the included GT sets set with regard

[1] https://swcregistry.io.

[2] https://dasp.co/.

to the three aspects accuracy, relevance, and provenance. For accuracy, we consider incompleteness, redundancy, and inconsistency; for relevance, we consider heterogeneity, amount of data, and timeliness; for provenance, we consider accessibility and trustworthiness.

2 Definition of Terms

To discuss the data, we use the following terms.

Property, Weakness, Vulnerability: Most contract properties addressed in datasets constitute program weaknesses, with a few exceptions like honeypots. In software engineering at large, vulnerabilities are weaknesses that can be actually exploited, while blockchain literature tends to use the two terms synonymously. Throughout the paper, we prefer the term *weakness*, and use *property* for general statements.

Judgment: If a property holds, the corresponding judgment is “positive”. If a property does not hold, the judgment is “negative”. If the assessment is inconclusive or does not make sense, the judgment is “not available” (n/a).

Assessment: a triple consisting of a contract, a single property, and a judgment of the latter in the context of the former.

Entry: smallest unit of a dataset according to its authors. Depending on the structure of the dataset, an entry consists of a single assessment or of multiple assessments pertaining to the same contract. We use the term mainly to relate to the original publication accompanying the dataset.

Contradiction: a group of two or more assessments for the same contract and property, but with conflicting judgments.

Duplicates: multiple assessments for the same contract and property with identical judgments.

3 Benchmark Sets with Ground Truth Data

In this section, we specify the selection of the benchmarks sets and give an overview of the contents in the included sets.

3.1 Selection of GT Sets

From the systematic literature review [11], where Rameder et al. identified benchmark sets of smart contracts for the quality assessment of approaches to weakness or vulnerability detection, we extracted all references that contain a ground truth. Moreover, for the years 2021 and 2022, we searched for further GT sets.

Inclusion Criteria. We include all sets that provide a ground truth by either manually checking the contracts or by generating them via deliberate and systematic bug injection.

Exclusion Criteria. We omit sets that reuse the samples of other sets without contributing assessments of their own. Moreover, we exclude sets having been assessed automatically, e.g. by combining the results of selected vulnerability detection tools by majority voting. While they may constitute interesting test data, they do not qualify as a ground truth.

3.2 Structure of the Included Sets

Table 1 lists the datasets that we selected as the basis of our work. They differ regarding the number of assessments per entry, the identification of contracts, the way assessments are specified, and the information provided per contract.

Identification: Usually, contracts are given either by a file with the Solidity source or by a chain address. Only one dataset specifies just an internal identifier, which in most cases contains an address.

Assessments: The majority of datasets provides the assessments in a structured form as `csv`, `json`, `xlsx` or `ods` files. Five datasets encode the weakness and partly also the judgment in the filepath or use prose.

Contract Information: The datasets may provide chain addresses, Solidity sources from Etherscan or elsewhere, deployment and/or runtime bytecodes.

Crafted and Wild Sets. Depending on the provenance of the contracts, we divide the datasets into two groups. The *wild* group comprises eight collections of contracts that have been deployed either on the main or a test chain, hence they all provide chain addresses or source code from Etherscan. The *crafted* sets contain at least some contracts that have not been deployed to a public chain[3]. One set has been obtained from the SWC registry, where it illustrates the SWC taxonomy. Two sets, JiuZhou and SBcurated, are related to tool evaluations. The set NotSoSmartContracts is intended for educational purposes, and the set SolidiFI was generated from Solidity sources by injecting seven types of bugs.

3.3 Summary of Assessments in the Included Sets

Table 1 gives an overview of the assessments in the sets. The first column contains a reference to the publication presenting the set, while the second one gives the number of entries. The subsequent columns quantify the assessments, specifying the total number as well as a breakdown by judgment type. The column for ignored assessments indicates the number of duplicate or contradicting assessments, as discussed in Sect. 5.2.

To compare the weaknesses covered by the sets, we map the individual assessments to the taxonomy provided by the SWC registry (Table 2). Section 5.3 discusses the mapping in detail. Properties not represented in the SWC registry

[3] The distinction between *crafted* and *wild* sets is not strict. Crafted sets may contain some contracts from public chains in modified or unmodified form.

Table 1. Included GT Sets.

	Set	reference	entries	assessments total	positive	negative	n/a	ignored	unmapped	weaknesses	SWC classes
wild datasets	CodeSmells	[3]	587	11740	2293	9073	374	1330	5870	20	10
	ContractFuzzer	[8]	379	379	379	0	0	4	0	7	7
	Doublade	[16]	319	319	152	167	0	40	0	5	5
	eThor	[13]	720	720	196	512	12	18	0	1	1
	EthRacer	[10]	127	127	69	47	11	16	0	2	1
	EverEvolvingG.	[19]	344	344	344	0	0	52	271	5	3
	NPChecker	[15]	50	250	28	222	0	31	0	5	5
	Zeus	[9]	1524	10533	2726	7807	0	3210	0	7	7
crafted	JiuZhou	[18]	168	168	68	100	0	3	39	53	33
	NotSoSmartC.		31	34	24	10	0	0	2	18	12
	SBcurated	[5]	143	145	145	0	0	16	0	10	16
	SolidiFI	[6]	350	350	350	0	0	7	0	7	7
	SWCregistry		117	117	76	41	0	1	0	33	33

remain unmapped, leading to unmapped assessments. The last two columns of Table 1 give the number of weaknesses as defined by the set and the number of covered SWC classes. When the number of weaknesses is larger than the number of SWC classes covered, it either means that there are unmapped assessments or that several weaknesses are mapped to the same SWC class.

4 Unified Ground Truth

In this section, we describe the process of merging the selected sets into a unified ground truth. We extract relevant data items, assign unique identifiers to the entries, repair mishaps, normalize the data to obtain a common format, add missing information from other data sources and investigate data variability.

4.1 Extracting Data from the Original Sets

For each repository selected (Sect. 3), we identify the parts pertaining to a ground truth, and use a Python script to extract relevant items. At a minimum, we need information to identify a contract, a property, and a corresponding judgment.

Most sets have not been designed for automated processing. They contain inconsistencies, errors, and information only intelligible to humans. We encountered numerous invalid Ethereum addresses, inconsistent spellings, invalid data formats, and wrong information (like bytecode not corresponding to the given source code). For the sake of transparency, we left the original sets unchanged and integrated the fixes into the Python scripts.

4.2 Completing the Data

To identify duplicate or contradicting assessments, and to arrive at a consolidated ground truth usable in different scenarios, each contract should be given by its source, deployment and runtime code as well as by its chain address (if deployed). Most repositories contain only some of this information. With the help of data from Ethereum's main chain and Etherscan's repository of source code, we were able to complete most missing data.

Contracts with Addresses:[4] We query the respective chain for the bytecodes, and Etherscan for the source code (if available).

Contracts with Source Code: We use the fingerprint of the source code to look it up in an internal database. If there is a match, we retrieve the deployment address and proceed as above. Otherwise, the source code can be compiled to obtain the corresponding bytecode. Given the variability of compilation, this step most likely will not result in code matching code obtained elsewhere, and is thus inferior when searching for duplicates.

Contracts with Bytecode: The contracts considered here all come with an address or some source code. However, to cross-check and to confirm guesses about the chain, we use fingerprints of any provided bytecode to look up public deployments. Moreover, we extract the runtime code from given deployment code.

Fingerprints. To detect identical contracts, we use fingerprints of the code. For source code, we eliminate comments and white space before computing the MD5 hash. A second type of fingerprint additionally eliminates `pragma solidity` statements prior to hashing. For bytecodes, we replace metadata sections inserted by the Solidity compiler with zeros before computing the MD5 hash.

4.3 Variability in the Unified GT Set

We portrait the variability with regard to the contract language (Solidity or EVM bytecode) as well as the range and distribution of Solidity versions and time of deployment.

Contract Identification. We need some reference to a contract, be it an address or a source file. Figure 1 depicts the number of entries in the unified GT set, for which we have an address, a source, both, or neither.

[4] Addresses by themselves are not sufficient to identify a contract. Apart from information about the chain, we also need the deployment time if the contract or an ancestor is the result of a CREATE2 operation. However, as the data in the repositories mostly predates the introduction of this operation, we encountered no contract of this type. Hence, for our purposes knowing the address and chain is sufficient. We use the block numbers of deployments only for analyzing changes over time.

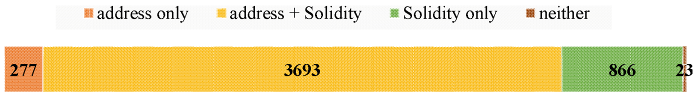

Fig. 1. Addresses (orange and yellow) and Solidity source files (yellow and green) in the entries in the unified GT set. (Color figure online)

In the unified set, there are 4 859 entries in total, of which 4 559 (93.8 %) come with a Solidity source and 3970 (81.7 %) with a deployment address. While 3693 (76.0 %) entries are associated with both, address and source, there are 866 (17.8 %) entries with a Solidity source only, and 277 (4.6 %), for which a source file is neither provided not can be retrieved. This concerns 28 entries in the set EverEvolvingGame, 131 in Zeus, and 118 in eThor. Moreover, 23 entries indicate neither an address nor a source file, but refer to a Solidity file without providing it (all in Zeus).

Chains. Entries with address refer to 2731 unique addresses, with the majority (2461) from the main chain, 268 from Ropsten, and one from Rinkeby. For one address in Zeus, we were not able to locate it on any public chain.

Solidity Versions. Solidity, the main programming language for smart contracts on Ethereum and beyond, has been evolving with several breaking changes so far. In the included sets, we see predominantly versions 0.4.x as depicted in the left part of Fig. 2. While the versions 0.4.x were current throughout 2017 up to early 2018, versions 0.8.x started December 2020 and are still current in mid 2023. The highest Solidity version in the GT sets is v0.6.4.

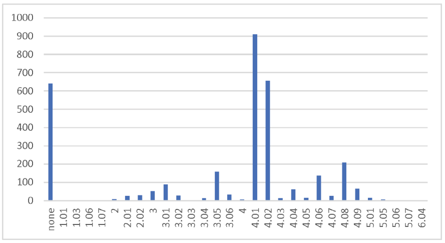

Fig. 2. Distribution of Solidity versions in the included GT sets.

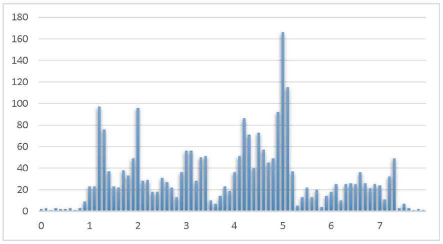

Fig. 3. Distribution of contract deployments/addresses in the included GT sets on a time line (in million blocks).

Deployment Blocks and Forks. To put the addresses into a temporal context, we depict the deployment block in Fig. 3. We count the deployments in bins of 100 000 blocks and depict them on a timeline of blocks (ticks per million blocks).

The latest block in the GT sets is 8 M, while by the end of 2022, the main chain was beyond block 16.3 M. The deployment block also indicates, which EVM opcodes (introduced by a regular fork) were available.[5] This information

[5] An important opcode change occurred at block 7.28 M with the introduction of the shift operations, which now appear in most contracts, and CREATE2. At block 9.069 M, SELFBALANCE and CHAINID got introduced, and at block 12.9 M BASEFEE.

may be critical if a detection tool was developed before a particualar opcode was introduced.

5 Consolidated Ground Truth

In this section, we describe the consolidation of the unified GT set. It consists of (i) identifying entries pertaining to the same contract, (ii) marking conflicts within sets, (iii) mapping all assessments to a common taxonomy, (iv) determining the overlaps between the included sets, and (v) analyzing disagreements between the sets.

5.1 Matching Contracts

To detect assessments referring to the same contract, we match the address and the fingerprints of the codes (cf. Sect. 4.2) according to the following considerations:

- Same address and chain means same contract, since none of the contracts in the sets was deployed via CREATE2.
- Most assessments are based on the Solidity source code. As the source usually specifies the admissible compiler versions (except when the missing directive is actually the weakness), the semantics of the program is fixed. So, if two source codes have identical fingerprints and the names of the contracts under consideration are the same, the assessments refer to the same contract.
- Assessments referring to deployment bytecodes with the same fingerprint can be considered as assessing the same contract, unless the checked property is tied to Solidity (like inheritance issues). For the SWC classes, Table 2 indicates the visibility of the weakness by a checkmark in the last column.
- Assessments referring to runtime codes with the same fingerprint are comparable only if the checked property is guaranteed to be detectable in this part of the code. Typically, this holds for weaknesses related to the contract being called by an adversary. For the SWC classes, Table 2 indicates the visibility of a weakness in the runtime code by a non-parenthesized checkmark in the last column. A checkmark in parentheses indicates that the weakness may occur in the constructor and thus is not necessarily detectable in the runtime code.

5.2 Assessments Excluded from the Consolidated Set

For obvious reasons, we ignore assessments where either the judgment is n/a, or where the object of the assessment is ill defined. The first condition affects 397 assessments, mostly from the set CodeSmells. The second one eliminates 153 assessments from the Zeus set, as some contract identifiers do not allow us to extract a valid chain address, and the set does not provide further information.

It is well known that contracts like wallets or tokens have been deployed identically numerous times. Often, this fact is not taken into account when collecting contract samples, such that the same contract may end up in a set multiple

Table 2. Coverage of SWC Classes in the Consolidated GT Set.

SWC-id	#Sets	pos.	neg.	Weakness	Bytecode
100	1	1	1	Function Default Visibility	
101	7	807	322	Integer Overflow and Underflow	(✓)
102	1	1	0	Outdated Compiler Version	
103	3	513	46	Floating Pragma	
104	10	426	1402	Unchecked Call Return Value	(✓)
105	4	62	5	Unprotected Ether Withdrawal	✓
106	3	7	3	Unprotected SELFDESTRUCT	✓
107	12	354	2087	Reentrancy	✓
108	2	3	1	State Variable Default Visibility	
109	3	6	3	Uninitialized Storage Pointer	
110	2	15	8	Assert Violation	(✓)
111	2	2	2	Use of Deprecated Solidity Functions	
112	4	33	6	Delegatecall to Untrusted Callee	(✓)
113	8	331	1359	DoS with Failed Call	(✓)
114	8	535	726	Transaction Order Dependence	✓
115	7	79	1619	Authorization through tx.origin	(✓)
116	5	174	1	Block values as a proxy for time	(✓)
117	2	3	2	Signature Malleability	(✓)
118	4	9	3	Incorrect Constructor Name	
119	3	4	3	Shadowing State Variables	
120	8	315	1423	Weak Sources of Randomness	(✓)
123	1	1	1	Requirement Violation	(✓)
124	4	10	4	Write to Arbitrary Storage Location	✓
125	2	2	2	Incorrect Inheritance Order	
127	2	2	1	Arbitrary Jump	✓
128	5	25	529	DoS With Block Gas Limit	(✓)
129	2	4	1	Typographical Error	
130	2	2	1	Right-To-Left-Override control character	
131	1	2	2	Presence of unused variables	(✓)
132	5	16	566	Unexpected Ether balance	(✓)
133	2	2	3	Hash Collisions	(✓)
134	2	18	0	Message call with hardcoded gas amount	(✓)
135	2	12	525	Code With No Effects	(✓)
136	2	3	3	Unencrypted Private Data On-Chain	
995	2	2	1	Short Address Attack	✓
996	3	51	1	Honey Pot	✓
997	3	86	530	Locked Ether	(✓)
999	1	3	7	Other Arithmetic Issue	(✓)

(✓) provided the weakness does not occur exclusively in the constructor times, albeit under different addresses. Therefore, we check the sets for multiple assessments of the same code, to find contradictions and duplicates.

Surprisingly, the Zeus set contains 18 contradictions already on the level of its own identifiers (meaning that the same identifier is listed multiple times, with diverging assessments) and 30 more when applying the criteria laid out in the last section. Moreover, we find 103 conflicts in the set CodeSmells, 6 in Doublade, and 3 in JiuZhou. These assessments are excluded from the consolidated set.

For duplicates, all but one assessment are redundant and can be ignored. We find duplicates in almost every set (the number in parentheses gives the ignored assessments): CodeSmells (853), ContactFuzzer (4), Doublade (34), eThor (6), EthRacer (5), EverEvolvingGame (52), NPChecker (31), SBcurated (16), SolidiFI (7), SWCregistry (1), and Zeus (3009).

For a summary of the exclusions, see the column 'ignored' in Table 1.

5.3 Mapping of Individual Assessments to a Common Taxonomy

To compare assessments in different sets, we map the properties of each set to classes of a suitable taxonomy. The SWC registry[6] provides such a widely used taxonomy with 37 weakness classes. Each has a numeric identifier, a title, a CWE parent and some code samples.

Coverage of SWC Classes. Table 2 shows how well the SWC classes are covered by positive and negative assessments, and how many sets contribute assessments to the class. Popular weaknesses with seven or more contributing GT sets are marked in gray. At the bottom, we add the classes 995–999 to account for weaknesses missing from the SWC registry.

Even after combining all GT sets into a unified ground truth, the coverage of the SWC classes remains highly uneven. This can be attributed to the intention behind most benchmark sets: to support the test of tools for automated vulnerability detection. And tools aim for "interesting" weaknesses.

Comparison of Weaknesses. It is intrinsically difficult to compare weaknesses across GT sets due to (i) vague or missing *definitions* of weaknesses, (ii) the unclear *relationship* between definitions, (iii) the ambiguous *mapping* of a weakness to a corresponding class, and (iv) heterogeneous *criteria for structuring* weaknesses, mixing cause and effect or different levels of the protocol/software stack. The definitions provided by GT authors are rarely a perfect match for a taxonomy. Therefore, when comparing weaknesses via a taxonomy, we have to check disagreements manually to distinguish mismatches of definitions from contradicting assessments.

5.4 Overlaps

To find disagreements between the cleaned GT sets, we first determine their overlap. For each pair of sets, Table 3 gives the number of non-ignored assessments that map to the same SWC class. The diagonal shows the total number

[6] https://swcregistry.io.

Table 3. Overlap of Mapped Assessments in the Consolidated GT Set.

Set	CodeSmells	ContractFuzzer	Doublade	eThor	EthRacer	EverEvolvingG	NPChecker	Zeus	JiuZhou	NotSoSmartC	SBcurated	SolidiFI	SWCregistry
CodeSmells	5300	6	4	26	0	0	14	145	0	1	0	0	0
ContractFuzzer	6	375	6	0	0	0	3	15	0	0	10	0	0
Doublade	4	6	279	2	0	0	2	10	0	0	7	0	0
eThor	26	0	2	702	0	0	25	691	0	0	0	0	0
EthRacer	0	0	0	0	111	0	0	5	0	0	0	0	0
EverEvolvingG.	0	0	0	0	0	65	0	0	0	0	0	0	0
NPChecker	14	3	2	25	0	0	219	128	0	0	0	0	0
Zeus	145	15	10	691	5	0	128	7323	0	0	1	0	0
JiuZhou	0	0	0	0	0	0	0	0	129	0	0	0	2
NotSoSmartC	1	0	0	0	0	0	0	0	0	32	7	0	0
SBcurated	0	10	7	0	0	0	0	1	0	7	129	0	31
SolidiFI	0	0	0	0	0	0	0	0	0	0	0	343	0
SWCregistry	0	0	0	0	0	0	0	0	2	0	31	0	116

of mapped assessments per GT set. The upper-left block relates *wild* sets, while the lower-right block concerns the *crafted* ones. As to be expected, there is more overlap within the *wild* group than within the *crafted* one or between the groups. SBcurated mixes crafted and wild contracts, with some crafted ones taken from the SWCregistry. Of 20 498 cleaned assessments, 18 409 appear in only one set, while 2 089 occur in two or more.

5.5 Disagreements and Errors

In Table 3, overlaps with disagreements are marked gray. Of the 2 098 overlapping assessments, 458 disagree with at least one other, involving eight GT sets and six SWC classes (Table 4).

The disagreements constitute an interesting area of investigation. While some disagreements are due to diverging definitions of weaknesses that were mapped to the same SWC class, quite a few turn out to be inconsistencies under the authors' original definitions. Table 4 summarizes the results of our manual evaluation. For each affected set, it gives the total number of assessments that disagree with an assessment of another set as well as a breakdown by SWC class. A table entry is marked red if our evaluation revealed assessment errors, giving also the number of such errors.

Table 4. Number of Disagreements in the Unified GT Set, with the Errors.

Dataset	total	104	107	113	114	120	997
CodeSmells	34	8	6 of 9	8	:	7 of 7	2 of 2
ContractFuzzer	13	7	:	:	:	4 of 4	2
Doublade	6	3	3 of 3	:	:	:	:
eThor	166	:	166	:	:	:	:
EthRacer	2	:	:	:	2 of 2	:	:
NotSoSmartC	1	:	1	:	:	:	:
NPChecker	25	3 of 6	1 of 7	7	1 of 2	2 of 3	:
Zeus	211	18	3 of 163	15	1 of 4	11	:

Table 5. Number of Manually Checked Assessments, with the Errors.

Set \ SWC	101	104	107	112	113	114	115	120	997
CodeSmells	:	1	3	:	3	:	:	:	2 of 7
ContractFuzzer	:	10	3	4 of 11	:	:	:	4	9
Doublade	:	1	2	:	:	:	1	:	:
NPChecker	:	1 of 4	:	:	:	:	:	:	:
SBcurated	:	1	2	:	:	:	:	:	:
Zeus	4 of 6	7	:	:	1	4	:	:	:

Since reentrancy is the most popular weakness, it appears in 12 GT sets and gives rise to most overlaps: of the 2 098 overlapping assessments, 1 480 pertain to reentrancy (SWC 107). Thus, it is not surprising that we observe the highest number of disagreements (182) and errors (13) for reentrancy.

With 42 disagreements, SWC 104 is second. However, we identified only three errors, with the other disagreements resulting from diverging definitions. While SWC 113 shows no errors, half of the disagreements for SWC 114, 120 and 997 are errors.

To gain further insights into the quality of the assessments, we randomly select 80 assessments from the consolidated ground truth, in order to manually check them. Table 5 shows, for each GT set and SWC class, the number of checked assessments as well as the number of errors.

6 Discussion

6.1 Data Quality

To assess the data quality of the GT sets along the dimensions proposed by Bosu et al. [2], we define scores for each criterion as specified in Table 6. The resulting overview of the data quality is shown in Table 7.

Table 6. Criteria for Data Quality Assessment.

	Aspect	Score	Criterion
Accuracy	completeness	+	source and bytecode are provided (and addresses if any)
		∘	some source and bytecodes provided
		−	no code files are provided
	irredundancy	+	duplicates [%] ≤ 1
		∘	$1 >$ duplicates [%] < 10
		−	duplicates [%] ≥ 10 (without bytecode)
	consistency	−	the dataset shows contradictions and conflicts
		∘	Tables 4 or 5 show inconsistent or incorrect assessments
		+	otherwise
Relevance	heterogeneity		Sections 4.3 and 6.1 detail the heterogeneity of the GT sets
		−	All sets show low heterogeneity.
	data quantity		Table 1 lists the number of assessments A and weaknesses W
		+	$A > 1000 \wedge W > 5$
		−	$A < 500 \wedge W < 5$
		∘	otherwise
	timeliness	−	All sets lack recent contracts (Sect. 4.3).

Table 7. Data Quality of Ground Truth Sets.

	Set	wild								crafted					
Quality Aspect		CodeSmells	ContractFuzzer	Doublade	eThor	EthRacer	EverEvolvingG	NPChecker	Zeus	JiuZhou	NotSoSmartC	SBcurated	SolidiFI	SWCregistry	Consolidated GT
Accuracy	completeness	−	+	∘	+	−	−	−	−	∘	∘	∘	∘	+	+
	irreduncancy	∘	+	∘	+	∘	−	−	−	+	+	+	∘	+	+
	consistency	−	∘	−	+	∘	+	∘	−	−	+	+	+	+	∘
Relevance	heterogeneity	−	−	−	−	−	−	−	−	∘	−	∘	−	∘	∘
	data quantity	+	∘	∘	∘	−	∘	∘	+	∘	∘	∘	∘	∘	+
	timeliness	−	−	−	−	−	−	−	−	−	−	−	−	−	−

Accuracy. All sets provide a minimum of data, but we had to complete the data of about two thirds of the sets. Redundancy exists in many *wild* GT sets, to varying degrees. The main concern regards inconsistencies – the key aspect of a GT – which we encountered in six *wild* GT sets. We improved the data quality (i) by data completion, (ii) by eliminating redundant and contradictory assessments within sets, and (iii) by resolving disagreements between sets. Thus, we could increase the accuracy in the consolidated GT set in all aspects. However, random inspections revealed further inconsistencies; the overall accuracy would benefit from further checks.

Relevance. The GT sets mostly lack heterogeneity, often provide a smallish amount of data, and above all lack recent data. By merging 13 GT sets, we could improve the amount of data (number of positive and especially negative samples) and some aspects of heterogeneity, like the number of weaknesses covered. However, there is still a bias towards a small range of Solidity versions, deployments between 2016 and 2018, source code that has been published for some reason on Etherscan, and popular vulnerabilities.

6.2 Related Work

Each of the thirteen original datasets can be regarded as distantly related work; see Sect. 3 for a description. Concerning the construction of a unified GT set, we only find AutoMESC [14]. In this work, Soud et al. choose five source code datasets [4,7,12,17,18] that address 10 vulnerabilities, which are detected by one or more of the tools HoneyBadger, Mythril, Maian, Osiris, Slither, SmartCheck, Solhint. They apply as inclusion criteria: recent (up to three years old), public, Ethereum, corresponding publication or GitHub repo; and exclude commercial and competition datasets, and sets that just provide one sample per vulnerability. For unification, they use a file-ID per contract (without checking for non-obvious duplicates). For consolidation (identifying duplicates, mapping the assessments to a common taxonomy, and resolving contradictory assessments), they discard the original classification, and replace it with a simple majority vote of the seven selected tools if those claim to detect the weakness (after mapping the tool findings to a common taxonomy). They claim that there is neither redundancy nor inconsistency in the five datasets included.

6.3 Challenges in Identifying Weaknesses

Ambiguous Definitions of Weaknesses. Hardly any weakness possesses a commonly accepted, precise definition. As a consequence, seemingly contradictory assessments of a contract by different datasets may actually result from applying subtly different definitions.

Weakness vs. Vulnerability. There is no agreement among dataset authors whether to aim for exploitable or potential issues.

Intended Purpose. The verdict on whether a weakness is considered a vulnerability also depends on the purpose of a contract. An apparent weakness may be actually the intended behavior of the contract (e.g. a faucet that “leaks” Ether).

Contracts in Isolation. The included datasets consider single-contract weaknesses only (discounting the attack contract). However, vulnerabilities may be the result of several interacting contracts. A single contract may not provide sufficient context to be classified as vulnerable on its own.

6.4 Reservations About Majority Voting

Due to the scarcity of GT data, some authors resort to pseudo-GT data. They run several vulnerability detection tools on selected contracts and obtain the judgment by comparing the number of positive results to a threshold. This approach is debatable for the following reasons.

Weakness vs. Vulnerability. Most tools detect code patterns that indicate a weakness, regardless of whether it can be actually exploited. Hence, false positives (and, to a lesser extent, false negatives) are rather the norm than the exception. Thus, majority vote may turn false positives into a positive assessment.

Tool Genealogy. Tools form families by being derived from common ancestors (like Oyente), by implementing the same approach (like symbolic execution, taint analysis, or fuzzing), or by relying on the same basic components (like GigaHorse, Rattle, Z3, or Soufflé). Related tools may misjudge a contract in a similar way and outnumber tools with the correct result.

Diverging Definitions of Weaknesses. Even if labeled the same, the weaknesses detected by any two tools are not quite the same. Rather, we are faced with tools voting on a weakness that is more or less similar to what they can detect.

7 Conclusion

Publicly available ground truth data for smart contract weaknesses is scarce, but much needed. Our consolidated ground truth is an appreciation of the commendable efforts by others and hopefully renders the included GT sets more usable to the community. The consolidated ground truth described in this paper is available from http://github.com/gsalzer/cgt For an extended version of this paper, see [1].

Future Work. *Granularity.* This unified and consolidated GT set is constructed on contract level. Information on the location of weaknesses within the contracts, like the line number in the source or the offset in the byte code, is available only for two small datasets, and was omitted here. *Severity Level.* Assigning a severity level to a weakness would further improve the GT set, but is a difficult topic on its own. *Updates* are important. We invite everyone to contribute by adding GT collections, taxonomies, levels of granularity or severity, proofs and exploits.

References

1. di Angelo, M., Salzer, G.: Consolidation of ground truth sets for weakness detection in smart contracts. arXiv preprint 2304.11624 (2023). https://doi.org/10.48550/arXiv.2304.11624
2. Bosu, M.F., MacDonell, S.G.: A taxonomy of data quality challenges in empirical software engineering. In: 2013 22nd Australian Software Engineering Conference, pp. 97–106. IEEE (2013). https://doi.org/10.1109/ASWEC.2013.21
3. Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: Defining smart contract defects on ethereum. IEEE Trans. Softw. Eng. (2020). https://doi.org/10.1109/TSE.2020.2989002
4. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 530–541. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380364
5. Ferreira, J.F., Cruz, P., Durieux, T., Abreu, R.: SmartBugs: a framework to analyze solidity smart contracts. In: 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020), pp. 1349–1352. ACM (2020). https://doi.org/10.1145/3324884.3415298
6. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 415–427. ISSTA 2020, Association for Computing Machinery (2020). https://doi.org/10.1145/3395363.3397385
7. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: thorough, declarative decompilation of smart contracts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1176–1186. IEEE (2019). https://doi.org/10.1109/ICSE.2019.00120
8. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 259–269. ASE 2018, Association for Computing Machinery (2018). https://doi.org/10.1145/3238147.3238177
9. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: NDSS Symposion. NDSS, Internet Society (2018). https://doi.org/10.14722/ndss.2018.23082
10. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 363–373. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330560
11. Rameder, H., Angelo, M.D., Salzer, G.: Review of automated vulnerability analysis of smart contracts on ethereum. Front. Blockchain - Smart Contracts (2022). https://doi.org/10.3389/fbloc.2022.814977
12. Ren, M., et al.: Empirical evaluation of smart contract testing: what is the best choice? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 566–579 (2021). https://doi.org/10.1145/3460319.3464837
13. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: EThor: practical and provably sound static analysis of ethereum smart contracts. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 621–640 (2020). https://doi.org/10.1145/3372297.3417250

14. Soud, M., Qasse, I., Liebel, G., Hamdaqa, M.: Automesc: automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. arXiv preprint arXiv:2212.10660 (2022). https://doi.org/10.48550/arXiv.2212.10660
15. Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in ethereum smart contracts. Proc. ACM Program. Lang. (PACMPL) **3**(189), 1–29 (2019). https://doi.org/10.1145/3360615
16. Xue, Y., et al.: Doublade: unknown vulnerability detection in smart contracts via abstract signature matching and refined detection rules. arXiv preprint arXiv:1912.04466 (2019). https://doi.org/10.48550/arXiv.1912.04466
17. Yashavant, C.S., Kumar, S., Karkare, A.: Scrawld: a dataset of real world ethereum smart contracts labelled with vulnerabilities. arXiv preprint arXiv:2202.11409 (2022). https://doi.org/10.48550/arXiv.2202.11409
18. Zhang, P., Xiao, F., Luo, X.: A framework and dataset for bugs in ethereum smart contracts. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 139–150. ICSME 2020, IEEE (2020). https://doi.org/10.1109/icsme46990.2020.00023
19. Zhou, S., Yang, Z., Xiang, J., Cao, Y., Yang, Z., Zhang, Y.: An ever-evolving game: evaluation of real-world attacks and defenses in ethereum ecosystem. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2793–2810. USENIX Security 2020, USENIX Association (2020). https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-shunfan