Broken Agreement: The Evolution of Solidity Error Handling

Charalambos Mitropoulos*
Technical University of Crete
Greece
cmitropoulos@isc.tuc.gr

Maria Kechagia*
University College London
United Kingdom
m.kechagia@ucl.ac.uk

Chrysostomos Maschas
GRNET
Greece
chrysom@noc.grnet.gr

Sotirios Ioannidis
Technical University of Crete
Greece
sioannidis@tuc.gr

Federica Sarro
University College London
United Kingdom
f.sarro@ucl.ac.uk

Dimitrios Mitropoulos
University of Athens
Greece
dimitro@ba.uoa.gr

Abstract

Background. A smart contract is a computer program enclosing the terms of a legal agreement between two or more parties which is automatically verified and executed via a computer network called blockchain. Once a smart contract transaction is completed the blockchain is updated and the transaction cannot be changed anymore. This implies that any error codified in the smart contract program cannot be rectified. Therefore, it is of vital importance that developers of smart contracts properly exploit error handling to prevent issues during and after the contract execution. Existing programming languages for smart contracts, support developers in this task by providing a set of Error Handling (EH) features. However, it is unclear the extent to which developers effectively use EH in practice. **Aims.** Our work aims to fill this gap by empirically investigating the state of practice on the adoption of EH features of one of the most popular programming languages for smart contracts, namely Solidity. **Method.** We empirically analyse the usage of EH features in 283K unique open-source Solidity smart contracts for the Ethereum blockchain. **Results.** Our analysis of the documentation of the different versions of Solidity coupled with the empirical evaluation of the EH uses and misuses found in real-word smart contracts, indicate that, among other things, Solidity EH features have been changing frequently across versions, and that the adoption of most of the Solidity EH features has been limited in practice. However, we observe an upward trend in the usage of the `require` EH feature, which is specifically designed for smart contract development. **Conclusions.** The insights from our study could help developers improve their EH practice as well as designers of smart contract programming languages to equip their language with appropriate EH features.

CCS Concepts

• **Software and its engineering** → **Error handling and recovery**; **Software evolution**.

Keywords

Solidity, smart contracts, error handling, software evolution

ACM Reference Format:
Charalambos Mitropoulos, Maria Kechagia, Chrysostomos Maschas, Sotirios Ioannidis, Federica Sarro, and Dimitrios Mitropoulos. 2024. Broken Agreement: The Evolution of Solidity Error Handling. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3674805.3686686

1 Introduction

Smart contracts [19, 30, 48] are computer programs stored on a *blockchain* (i.e., a system maintaining a record of *transactions* across computers linked in a peer-to-peer network [40, 44]) that can be used for automating the execution of transactions between different parties. Examples of transactions include triggering a payment or a service delivery, registering a vehicle, or issuing a ticket.

Solidity [17] is a recent object-oriented programming language, released in 2014, for developing smart contracts that run on blockchain platforms such as *Ethereum* [14]. Ethereum is one of the largest and most popular decentralised platforms where 1M transactions take place everyday [13, 25].

As general-purpose programming languages provide the developers with some sort of *error-handling* to help them handle unexpected errors that may manifest at run-time [45, 59, 64, 66], Solidity, also, provides a number of *error-handling* (EH) *features* [12], e.g., to handle calls to other smart contracts, which can be exposed to potential runtime errors e.g., attacks due to a vulnerability. Such features include standards like `try-catch` and `assert`, as well as features developed specifically for Solidity, i.e., `require` and `revert`.

Despite Solidity provide several EH features, it seems that developers do not fully understand how they work and often neglect their usage[41]. The Nomad bridge attack [1], which costed millions of dollars, is just one of the most recent examples of how a correct usage of EH features can prevent critical financial losses. In fact, this attack was due to a missing check for a zero `address` (`0x00`). Without this check, the contract would be marked a zero `address` as a valid `address` for incoming messages. Thus, attackers were able to perform malicious, yet valid transactions by using the `0x00` `address` as a stepping stone. The Nomad bridge attack could have been avoided if the developers had used the `require` EH feature provided by Solidity (as we will explain later on, `require` can be used to verify external inputs before execution).

---

[*] The first two authors contributed equally to this work.

ESEM '24, October 24–25, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-1047-6/24/10
https://doi.org/10.1145/3674805.3686686

Amann et al. [20, 60] consider the coding situation where *explicit* error handling is missed (similar to the case of the Nomad attack), as an EH *misuse* i.e., a violation of the specification usage of a programming language's element, e.g., an error-handling feature. Due to the serious financial and legal implications of errors in the program logic of smart contracts (e.g. missing error handling), the reliable execution of a contract through the *correct* usage of EH features should be a top priority for developers of smart contracts.

Even though there are several studies that examine the EH usage and its impact for general-purposes programming languages [23, 24, 43, 50, 52, 59], this is not the case for Solidity. Moreover, Solidity is a relatively new programming language, and, as such, its characteristics and features evolve on a daily basis. Thus, focusing on the evolution of its EH features and corresponding documentation would provide Solidity's designers and smart contract's developers with important insights. While there are several empirical studies related to Solidity, focusing, for instance, on the use of inline assembly [28], the performance of program-analysis tools [34], code reuse [32], and library misuses [41], to the best of our knowledge, there is no empirical study that examines the evolution of Solidity EH features.

Our work aims to fill this gap: We carry out the first large-scale empirical study (involving 283K unique open-source Solidity smart contracts) aiming at understanding how developers use Solidity EH features over time and making suggestions to facilitate the features' correct design and use. Specifically, we study the evolution of the provisioning of EH features as new Solidity versions are released (**RQ1**), the frequency and category of EH features used by developers in their contracts (**RQ2**), and how this usage evolve over time (**RQ3**). Moreover, we analyse whether developers misuse EH features, categorise potential misuses, examine their impact, and quantify their occurrence (**RQ4**) and evolution over time (**RQ5**).

Overall, our findings show that substantial changes in Solidity EH features include the deprecation of `throw` early on, and the introduction of `require` and `try-catch` at different points in time (**RQ1**). The most used EH feature is `require` (83.15%), while the least used EH feature is `assert` (3.82%) (**RQ2**). Furthermore, `require` has the highest usage increase across Solidity versions and over the years. When `try-catch` was introduced in Solidity, the number of its usage increased, instantly, becoming equal to those of `revert` and `assert` (**RQ3**). Popular misuses involve missing EH features to check if either an `address` type (i.e., the `address` of a block) is valid, or if the call performed to an external contract was successful (**RQ4**). Moreover, according to our analysis of the Solidity documentation, we found that it does not contain enough examples illustrating the different coding situations where EH features should be used. This means that the Solidity documentation does not clarify the actual impact in the case of a misuse. Furthermore, we observe that the highest number of misuses over time involves the absence of EH for validating calls to external contracts, and checking the reliability of blockchain addresses (**RQ5**). We also observe a positive increase, over time, in the adoption of some specific EH features. Notably, an upward trend (> 60%) in the usage of Solidity-tailored EH features (i.e., `require`) suggests that designers of modern programming languages may want to consider devising EH features that are more specific to each language's purposes.

In summary, we present the first empirical study analysing both quantitative and qualitative aspects of the usage of Error Handling in Solidity over time, by analysing the largest number of real-world smart contracts to date and making the following contributions:
Evolution. We identify significant changes in EH features offered by Solidity over time and the corresponding developers' uptake and usage of such features.
Misuses. We identify seven categories of misuses regarding Solidity EH, providing representative examples, and discussing their implications for the users.
Suggestions. We suggest improvements for Solidity EH features and their documentation, based on the empirical evidence sought.
Open Science. We make publicly available the dataset we curated and the tool we created to analyse the data (SolBench) [2], in order to allow for replication and extension of our study.

2 Background

Error-Handling in Solidity. Solidity is based on `solc`, the *standard* Solidity compiler, which counts ~100 releases since 2015 [17]. `solc` offers several mechanisms including EH features and the SMTChecker, a built-in *formal verification module* [16]. According to Solidity's API reference documentation (hereafter, *Solidity documentation*) [12],[1] Solidity uses state-reverting exceptions to handle potential runtime errors. When an exception manifests in a sub-call, it is propagated unless it is caught via error handling.

Two EH features tailored to Solidity are `require` and `revert`. `require` checks for programming conditions, and throws an exception when particular conditions are not met. `revert` has the same semantics as the `throw` keyword used in older Solidity versions (it was removed in version 0.5.0 as we will see later in the paper), and in other programming languages such as Java [3]. If `revert` is triggered, an exception is thrown, along with the return of *gas* (i.e., the fee required to perform a transaction on the Ethereum blockchain), and reverts to its original state. `solc` also supports two EH features inspired by other programming languages (e.g. Java), namely: `try-catch` and `assert`. A `try` statement can be used to define a block of code to be tested for errors, while it is being executed. `catch` includes a block of code to be executed if the error that occurred was in the corresponding `try` block. `assert` can be employed to check for specific conditions and if the conditions are not met it throws an exception.

Error-Handling Misuses. The usage specification of the Solidity EH features. An EH misuse occurs when the developers of smart contracts violate the usage specification of the EH features. Such misuses may affect the reliable execution and the security of a smart contract. Specifically, if developers ignore the Solidity EH features, particular exceptions can manifest causing detrimental effects. EH misuses may also introduce security vulnerabilities [4, 5, 29, 55] that can lead even to a DoS (Denial of Service) attack [6]. For instance, recall the Nomad Bridge attack [1] (discussed in Section 1) that led to a $190M loss. This attack was based on a missing `address`-zero check, which should have been performed via error handling.

---

[1] We refer to Solidity versions ≤ `0.8.19` since `0.8.19` is the latest version found available at the time of our study.

According to a recent taxonomy of API misuses [20],[2] there are two categories referring to EH misuses: (1) missing usage of error handling and (2) redundant usage of error handling. In this study, we examine coding situations where error handling is missing (hereafter, EH *misuses*). Notably, the aforementioned work indicates that the absence of EH usage can have a greater negative impact on the functionality of a program than the redundant EH usage. Even though the work of Amman et al. [20] refers to API-misuse categories for Java APIs, we argue that the identified categories are generic enough to be applied to APIs for different programming languages, including Solidity.

Listing 1 presents a code excerpt, where EH is missing, i.e., the developer should have used `require` to check that the `account` (see the first argument in line 2) is not a zero `address` [12]. Recall that a similar check was missing in the Nomad bridge attack [1].

Listing 1: Missing zero-`address` check via `require`.

```
1   //contract ScalpexToken
2   function _burn(address account, uint256 amount) internal virtual {
3       _beforeTokenTransfer(account, address(0), amount);
4       _balances[account] = _balances[account].sub(amount, "ERC20:burn␣amount␣exceeds␣balance")
            ;
5       _totalSupply = _totalSupply.sub(amount);
6       emit Transfer(account, address(0), amount);
7   }
```

3 Empirical Study Design

3.1 Research Questions

Our work aims at investigating two main aspects of Solidity EH: How the EH features have been provided by Solidity over time, and how the EH features have been used by developers in practice.

Our first research question focuses on the manual analysis of the Solidity documentation of 102 releases to investigate the provisioning of EH features by Solidity and their characteristics over time, including observing newly proposed or deprecated EH features. Thus, we ask: **RQ1: How do the provisioning of** EH **features evolve as new Solidity versions are released?**

Next, we investigate two research questions that aim at empirically investigating the extent to which developers use error-handling in Solidity smart contracts that are deployed in production by using the largest real-word Solidity dataset to date.

We identify the popularity of all EH features. For the ones that are scarcely used by developers, we investigate the possible reasons for it. To this end we ask: **RQ2: How frequently do developers use each of the Solidity** EH **features?**

Once we assess the use of Solidity EH features in practice, we dive deeper to identify trends and observe error-handling usages over time, e.g. before and after the introduction of important Solidity modules. This analysis will enable us, inter-alia, to understand whether developers follow the upgrades introduced in new Solidity versions and how these changes impact the usage of EH features. Thus we ask: **RQ3: How often do developers use Solidity** EH **features over time?**

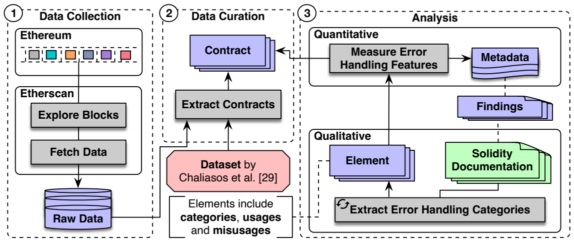

Figure 1: Our method to analyse the evolution of Solidity EH.

We also aim to elicit and categorise existing EH misuses and their potential impact. Furthermore, we intend to examine how often developers fail to use EH features when they should. Based on our results, we will be able to make suggestions to support developers in the correct usage of EH features. Thus we ask: **RQ4: What are the types of Solidity** EH **misuses and what is the overall frequency of each type?**

Finally we focus on the evolution of EH misuses over time as this would enable us to identify potential trends (e.g., a sudden increase) and correlate them with the corresponding Solidity release and given EH feature. This motivates our last question: **RQ5: How do Solidity** EH **misuses evolve?**

3.2 Methodology Overview

To address the research questions presented in Section 3.1, we carry out a large-scale empirical study consisting of three main steps: ① collecting raw smart contract data; ② processing this data and consolidating it together with a dataset publicly released in previous work; ③ designing and carrying out a quantitative analysis of smart contract usage over time as well as a qualitative analysis of Solidity documentation related to the various released versions, and the formulation of heuristics needed to identify EH uses and misuses. Figure 1 gives an overview of our methodology.

The first two steps are necessary to curate the largest dataset of Solidity smart contract to date, covering a wide period of time (i.e., from August 2015 to April 2023). We discuss the details of our data collection and consolidation in Section 3.3.

To answer RQ1, we thoroughly study the documentation of the various Solidity releases to extract information about error-handling, e.g., related code constructs, categories, and how developers can use it. Furthermore, through this qualitative analysis, we form the basis for answering the other four research questions by examining correct EH usages or lack thereof. We describe the protocol we follow to study the Solidity documentation in Section 3.4.

Furthermore, we build on the findings of our qualitative analysis to categorise EH usages in Solidity and to devise a set of heuristics, which are based on an extendable grammar. We describe the inferred EH categories and heuristics in Sections 3.5 and 3.6, respectively. To answer RQ2, we applied these heuristic rules to our dataset and calculated the frequencies of EH usage categories.

To answer RQ3, we examined the evolution of EH usages in Solidity across years, across versions, and the growth rate. We also analyse the extent to which developers respond to the evolution of EH features over time.

In order to answer RQ4 and RQ5, we use the heuristics to detect EH misuses (as we explain in Section 3.6). Moreover, we identify

---

[2] An Application Programming Interface (API) can be considered as a publicly available bundle of interfaces, classes, and methods, that client programs can call or implement. An API *usage* refers to any call of one or more methods of either old or new versions of an API, i.e., $API = [m1, ..., mk]$ [39]. An API misuse is a violation of the usage specification of that API.

Table 1: Descriptive statistics of our dataset.

Start Block	47,205 (7th of August, 2015)
End Block	16,993,877 (8th of April, 2023)
Total Source files	2M
Total Unique Source files	1.7M
Total Number of Functions	168M
Total Number of Modifiers	4.03M
Total Number of Constructors	672K
Total Unique Smart Contracts	**283K**
Contracts with at least one EH feature	215.8K
Average LOC per *.sol* file	105.1
LOC	51.2M

the potential effects of such misuses by exploiting the Solidity documentation. This analysis allowed us to examine the frequencies of EH misuses and highlight their potential impact (RQ4), and the EH misuse trends and growths over time (RQ5).

For those RQs that are related to the evolution over time, i.e., RQ3 and RQ5, we also use statistical inference tests to check whether the results obtained are statistically significant. Specifically, we use the Wilcoxon test [33] since our data is not normally distributed and set the significance level $\alpha = 0.05\%$.

3.3 Dataset

We curated a dataset containing 283K unique smart contracts. This dataset combines the most recent and largest public dataset of Solidity smart contracts collected from August 2015 to March 2022 by Chaliasos et al. [28] and the smart contracts that we collected from April 2022 to April 2023 via Etherscan REST API.

Data Collection via Etherscan REST API. For the data collection phase, we use the REST API of *Etherscan* [15], a well-established block explorer for Ethereum. We identify smart contracts from Etherscan for a specific period of time: from April 2022 to April 2023. To do so, we first examine raw data containing all the addresses of verified contracts. Then, we store their addresses and download their corresponding code. Note that not all developers make their smart contracts' source code publicly available. For our analysis, we only retain open-source smart contracts. Further, when a smart contract is downloaded, it is accompanied by various meta-data and additional files (including dates, information about the authors and more). We discard such files which are irrelevant to our study, and store only the smart contract's source code. By applying the aforementioned workflow we are able to gather contracts that were published after the study of Chaliasos et al. [28].

Curation. From both datasets we made sure to filter out empty smart contracts (i.e., without source code) as they do not provide meaningful information for our study, and repeated source code files due to code reuse. We found that ~ 1 million smart contracts from the data collected by Chaliasos et al. [28] were empty, and therefore we excluded them from our analysis. Note that our data do not contain any empty contract as they were automatically excluded during the data collection phase. We observe that reused code can stem from two sources: ① imported third-party libraries and ② contracts that include other pre-existing contracts. In both cases, we automatically detect reused code through an automated process. For ①, we detect whether the code of these libraries is incorporated into the code of a contract or it is included in the directory of the contracts. Consider that we analyse the library code only the first time we detect it. Namely, if the library is already present as an individual file in the directory, and we have already reviewed it, we skip it. For ②, we detect whether a contract previously examined is included in the directory of the current contract under examination. If so, then, we skip it. Using the above procedures, we effectively eliminated 300K files of reused code across both datasets.

All smart contracts use `solc`[3] versions from `v0.1.2` to `v0.8.19`. Since the range of `solc` versions which are compatible with a given smart contract are defined in the first lines of the contract itself by the keyword (`pragma solidity <version>`), we detect `solc` versions by examining these lines. Also, every smart contract has a timestamp, which indicates when the contract was actually deployed on Etherscan. This timestamp is stored in the contract's meta-data under the keyword `txhash`. This way, we can match the contracts to a given Solidity release. We consider Solidity versions from `v0.1.2` on-wards, because earlier versions do not have an accompanying documentation. Version `v0.8.19` was the latest available at the time of our study.

Description. Table 1 shows the descriptive statistics of our final dataset, which contains a total of 283K unique smart contracts deployed between August 2015 and April 2023, for a total of 1.7 million unique source code files. It is worth noting that a single smart contract may encompass multiple source files with a *.sol* extension. On average, a contract consists of around 7.25 *.sol* files, with a median value of 6 files. Additionally, the average lines of code per *.sol* file is approximately 105.1. In total, the dataset contains approximately 283,000 unique smart contracts and 76% of these contracts include at least one EH feature.

3.4 Documentation Analysis Protocol

An important part of our methododology involves the analysis of the Solidity documentation. To this end, two authors independently examined: ⓐ the documentation of the latest Solidity version at the time of our study (i.e., `v0.8.19`) to identify elements such as usage categories for the EH features, and ⓑ the documentation of each `solc` version from `v0.1.2` to `v0.8.19` to record changes related to EH features.[4] Through ⓐ, we are able to answer RQ2 and RQ3 to RQ5. Furthermore, ⓑ helps us to answer RQ1, but it also contributes to the results of other RQs, as described in Section 4.

The authors discussed their observations and findings until they reached a consensus. The procedure was repeated three times. During these iterations, the authors revisited and updated their observations. Overall, there was no disagreement between the two authors. Moreover, the two authors presented their findings to two additional authors that ratified the resulting findings.

3.5 Error-Handling Usage Categories

Studying the Solidity documentation (`v0.8.19`, i.e., the latest available version at the time of our study) we identify the EH usage

---

[3] We use `solc` or Solidity, interchangeably, to refer to the versions considered in our study. Solidity is based on `solc`, and, thus, `solc` or Solidity follow the same versions.

[4] `v0.1.2` is the first `solc` release that includes meaningful documentation files.

categories as described in the following paragraphs. The description of the EH features is based on Solidity documentation.[5] We analyse the use of the different EH features over time in Section 4.1.

`require` can be used to evaluate (1) function arguments and (2) external calls that are used to call functions from other contracts. Consider the code excerpt in Listing 2. The `setTrustedMarket` function calls the external function `getMarketOwner` from the contract `marketRegistry`. In this case, `require` is used to test if the call works as it is supposed to.

Listing 2: Use of `require` in external call.

```
//contract: TellerV2Context
function setTrustedMarket(uint256 _mId, address _forwarder) external {
    require( marketRegistry.getMarketOwner(_mId) == _Sender(), "Caller␣must␣be␣the␣
            marketowner");
}
```

`try–catch` can be used to: (1) catch and handle failures from external calls, in the same way as `require` does, and (2) evaluate the creation of external contracts. The last case occurs when a contract creates an external contract and the developer has to be sure that the creation is done correctly.

`revert` can be called as (1) a function and as (2) a statement to throw an error. Consider the code excerpt in Listing 3. In that case, the developer defines its own error function called `OnlyOwner()`. If the `if` statement is `true`, the `revert` function will be called providing an error message defined by the developer. In the same manner, in the case of a `revert` statement, the error message is provided inside quoted marks: `revert("...")`.

Listing 3: Use of `revert` with custom error.

```
//contract: Strategy
error OnlyOwner();
function updateOperatorFilterRegistryAddress(address newRegistry) public virtual {
    if (msg.sender != owner()) {
        revert OnlyOwner();
    }
}
```

`assert` can be used to check specific coding situations listed in the Solidity documentation [7]. Each case has a corresponding error code. After analysing all situations, we grouped them into five categories. The following list mentions the codes of each category considered in our study: (1) arithmetic overflow/underflow (with error code `0x11`), (2) division by zero (`0x12`), (3) array operations (`0x22`, `0x31`, `0x32`, `0x41`), (4) program logic (`0x01`, `0x51`), and (5) `enum` type conversion (`0x21`). Note, that certain categories can be identified by the SMTChecker starting from compiler version `0.8.7` (see Section 4.1). An example belonging to the `enum` type conversion category, involves the code excerpt in Listing 4.

Listing 4: Use of the `enum` type conversion.

```
//contract: Hakiro
enum Step { Before, PublicSale, WhitelistSale, SoldOut, Reveal}
function setStep(uint _step) external onlyOwner {
    assert(sellingStep = Step(_step));
}
```

Here, the developer attempts to convert a `uint` value named `_step`, into the corresponding `Step` `enum` value. To ensure that the process is done correctly, the `assert` function is called. Consider that the `0x00` code (i.e., *"generic compiler inserted panics"*) is not assigned to any category, because it is too generic to classify.

$$
\begin{array}{rcl}
\langle u \in Usages \rangle & ::= & \mathit{eh(c.f\ com\ (t \mid f),\ t \mid er)} \mid \mathit{eh(t\ com\ t,\ t)} \mid \\
 & & \mathit{eh(t\ op\ t)} \mid \mathit{eh(er \mid t)} \mid \mathit{eh(arr.l\ com\ t)\ \&\ u} \mid \\
 & & \mathit{eh(e(t))} \mid \mathit{eh(arr.}\mathbf{pop()}\mathit{)} \mid \mathit{eh(}\mathbf{new}\ \mathit{c.f)} \mid \\
 & & \mathit{eh(t\ com\ t)} \\
\langle eh \in Features \rangle & ::= & \mathbf{require} \mid \mathbf{revert} \mid \mathbf{try} \mid \mathbf{assert} \\
\langle c \in SmartContracts \rangle & ::= & \mathit{is\ the\ set\ of\ available\ smart\ contracts} \\
\langle f \in Functions \rangle & ::= & \mathit{is\ the\ set\ of\ functions} \\
\langle e \in EnumType \rangle & ::= & \mathit{set\ of}\ \mathbf{enum}\ \mathit{types} \\
\langle er \rangle & ::= & \mathbf{Error} \mid \mathbf{Panic} \mid \mathit{CustomError} \\
\langle arr \in arrays \rangle & ::= & \mathbf{storage} \mid \mathbf{memory} \\
\langle l \rangle & ::= & \mathit{array\ length} \\
\langle t \in SolidityTypes \rangle & ::= & \mathbf{str} \mid \mathbf{uint} \mid \mathbf{int} \mid \mathbf{bool} \mid \mathbf{address} \\
\langle op \rangle & ::= & + \mid - \mid * \mid / \\
\langle com \rangle & ::= & < \mid <= \mid > \mid >= \mid == \mid != \mid !
\end{array}
$$

Figure 2: The abstract syntax of the heuristic rules that represent the usages of the different error-handling (EH) features, according to the Solidity documentation (v0.8.19).

3.6 Heuristic Rules

To automatically detect EH usages we define a set of heuristic rules. The rules are based on the aforementioned usage categorisation (see Section 3.5) and the examination of real-world Solidity smart contracts containing EH features. Furthermore, the rules are generic and can be easily expanded if a new pattern appears in newer Solidity versions. Similarly, one can remove deprecated patterns that Solidity developers may mark as obsolete in the future. Note that our heuristic rules can be also used to identify EH misuses, because the denial of a heuristic indicates a potential misuse.

Figure 2, presents the set of our heuristic rules. Each element of the *Usages* set, requires an EH feature from the *eh* set: $\{require, revert, try, assert\}$. When we detect one of the *Usages* patterns in a smart contract that exists in our dataset, it means that we identify a (correct) usage of an EH feature. Every rule in *Usages* corresponds to a usage category presented in Section 3.5. Consider the rule: *eh(t com t)*. This can lead to the detection of two patterns of a program logic check, namely: `require(a >= b)` | `assert(c == d)`. Furthermore, through *eh(t op t)*, we can detect both (1) overflows/underflows and (2) division by zero checks (e.g., `assert(a * b)` | `assert(a / b)`.

As an example, consider Listing 2, where an external call takes place. In this case, the heuristic rule that checks for the correct usage of the `require` feature is the following: *eh(c.f com (t | f), t | er)*. Specifically, the rule searches for a pattern where the output of an external function *c.f* (`getMarkeOwner` in our case) is compared to (*com*) either the output of another function, *f* (`_Sender()` in our example), or another type, *t* (e.g., a message in a string). After the comparison, another object (*t*) is expected, i.e., the exception thrown by the feature (in our case a `string`).

A misuse appears when a heuristic rule is violated. Consider a situation where an external function call (*c.f*) such as

---

[5] Note that in `v0.8.19`, the `throw` EH feature was deprecated. We counted any `if ... throw` occurrences, in our dataset, and found that `throw` is used in a very small number of contracts (∼ 1%). Thus, we decided to exclude `throw` from our analysis.

`marketRegistry.getMarketOwner` in Listing 2, is detected. In this case. if `require` is not used to check the call, a misuse is identified. In our supplementary material, we provide more details regarding the rules we use to identify misuses. To implement the heuristic rules, we use regular expressions to detect EH patterns. To validate these rules, we developed a script [8] that initially employs the built-in AST generator of the Solidity compiler (`solc`) to produce the AST for a specified smart contract, and subsequently identifies the EH features within the AST of that smart contract. By doing so, we can cross-check whether the EH patterns detected by the heuristic rules align with those from the AST of the smart contract.

4 Empirical Study Results

4.1 RQ1: Evolution of Solidity EH Features

Since Solidity inception, the total number of releases of the Solidity compiler `solc` is 102 (until `v0.8.19`). Based on our manual analysis of the 102 versions (see Section 3.4) we found that only 13 releases of `solc` include a substantial change in at least an EH feature. We also observed that median and average number of days between two releases is 28.50 and is 27.91, respectively. While, the median number of days between two releases that include a change in EH is 112.5 and the average is 179.08.

Table 2 summarises the changes to Solidity EH features introduced in each of the 13 releases, as unveiled by our manual analysis. We observe that up to `v0.4.10`, there are no EH features, except for the `throw` keyword, which was subsequently deprecated in `v0.4.13` [18]. At the same time, three new EH features were introduced, namely, `require`, `revert`, and `assert`. Subsequently, `v0.4.22` was released with a number of features that enable developers to specify messages in `require` and `revert`. This version was also equipped for the first time with the SMTChecker, a formal verification module of `solc`, which allows developers to automatically detect, for instance, arithmetic overflows at compile time. The SMTChecker could be enabled by developers in their contract via the `pragma` keyword. In `v0.6.0`, `try-catch` was introduced, and from `v0.6.9` onwards, the SMTChecker was updated to check for array-related actions, overflows and underflows. It is worth noting that until then, such checks could be performed via `require` and `assert`. `v0.8.0` introduced two main types of exceptions: `Panic` and `Error`. `Panic` exceptions are created by the compiler in certain situations [7], or they can be triggered by the `assert` function. An `Error` exception can be generated by either `require` or `revert`. Such checks are used to ensure valid conditions that cannot be detected until execution time. Furthermore, from `v0.8.1` onwards, developers were given the ability to use `Panic` and `Error` inside a `catch` statement. In `v0.8.4` the SMTChecker was enabled by default, and since `v0.8.7` several evaluations such as division by zero and arithmetic overflows are performed at compiler level.

Answer to RQ1: Substantial changes in Solidity EH features include the deprecation of `throw`, the introduction of `require` and `try-catch`, as well as the introduction of the SMTChecker that currently supports checks performed via `assert`. On average, changes in EH features happen once or twice per year at least.

Table 2: RQ1. Error-handling (EH) changes across versions.

Version	Date	EH Change
0.1.3	Sep 23, 2015	Introduction of `throw`
0.4.0	Sep 8, 2016	Specification of the compiler version via `pragma` Introduction of `require(condition)` and `assert(condition)`
0.4.10	Mar 15, 2017	Support of `revert()` as an `OPcode` to abort with rolling back, but not consuming all gas.
0.4.13	Jul 6, 2017	Deprecation of `throw()` due to `require()`, `assert()`, `revert()`
0.4.16	Aug 24, 2017	Automated support for checking overflows and ASSERT
0.4.22	Apr 17, 2018	Specification of Error in `require` and `revert`
0.6.0	Dec 18, 2019	Introduction of `try-catch`
0.6.9	Jun 4, 2020	SMTChecker supports `require` and `assert`
0.7.2	Sep 28, 2020	SMTChecker supports `revert()`
0.8.0	Dec 16, 2020	Introduction of `Panic(uint)` and `Error(string)`
0.8.1	Jan 27, 2021	Catch and decode `Panic(uint)` and `Error(string)` in `try-catch`
0.8.4	Apr 21, 2021	Start deprecating `pragma experimental` SMTChecker
0.8.7	Aug 11, 2021	Enabling SMTChecker to check for overflows / underflows Running SMTChecker by default

Table 3: RQ2. Appearances (#) and frequencies (%) of error-handling (EH) usage categories.

EH Features	Usage Categories	(#)	(%)	Total (%)
`require`	function arguments	38,059	56.35	83.15
	external calls	29,541	43.75	
`try-catch`	external calls	4,135	70.25	7.25
	external contract creation	1,756	29.83	
`revert`	functions	3,821	81.22	5.79
	statements	885	18.88	
`assert`	overflow / underflow	1,217	46.93	3.82
	division by zero	984	37.94	
	array operations	218	8.43	
	program logic	107	4.12	
	enum type conversion	67	2.58	

4.2 RQ2: Frequencies of EH Usages

Table 3 shows, for each EH feature, the frequency of each usage category identified. We observe that the most used EH feature is `require` (83.15%). Notably, `require` is introduced specifically for Solidity and is not supported by other programming languages. The remaining EH features, `assert`, `revert`, and `try-catch`, are less used, i.e., < 10% per feature.

We also analyse the context that each EH feature is used for (i.e., usage category). `require` is more used for evaluating function arguments (56.3%) than for external calls (43.7%). `try-catch` is mostly used to handle a potential failure in an external call (70.2%) and less used to check whether the creation of an external contract is successful or not. Furthermore, `revert` function is utilised more (81.2%) than the `revert` statement (18.8%). This is meaningful because developers can write specific messages (e.g., "Not enough funds") through `revert` functions, and facilitate debugging. Finally, `assert` is mostly used to identify potential arithmetic overflows and underflows (46.93%) and division by zero (37.94%). By contrast, `assert` is rarely used to check for valid array operations (8.43%), examine `enum` type conversions (2.58%), or evaluate program-specific conditions (4.12%).

Answer to RQ2: The most used EH feature is `require` (83.15%), while the least used is `assert` (3.82%). `try-catch` is mostly used to evaluate external calls (>70%). `require` is slightly more used for evaluating function arguments (56.3%) than for external calls (43.7%). `assert` is mostly used for checking overflows / underflows (>35%) and division by zero (>30%). `revert` is mostly used as a function (>80%).

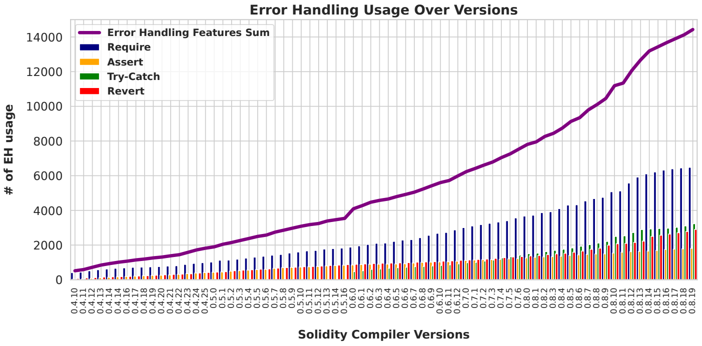

Figure 3: RQ3. Error-handling (EH) usage across versions.

Table 4: RQ3. Error-handling (EH) usage growth rates.

EH feature	Min	Max	Mean	Median	St. Dev.
`require`	0.163	3.938	1.894	2.085	1.187
`try-catch`	0.162	1.037	0.333	0.162	0.378
`revert`	0.220	0.927	0.513	0.490	0.224
`assert`	0.504	1.406	0.961	0.989	0.329

4.3 RQ3: Evolution of EH Usages In Practice

Figure 3 illustrates how many times each EH feature is used across the different Solidity versions together with their overall usage. (i.e., EH features' sum). If a smart contract with EH features can be compiled with `solc` ≥ `v0.4.10`, then we count each detected feature in this specific contract, for the whole range of these versions.

Overall, our findings indicate that developers seem to use all features more as time goes by. In addition, `require` is the most utilized EH feature across all Solidity versions. An interesting observation involves `try-catch`. The feature is introduced in `v0.6.0` (see Table 2), and its usage is gradually increasing until `v0.7.4` (appearing 1,275 times in this version) and eventually takes the lead from all the other EH features, except for `require`.

Figure 4, illustrates the frequencies of EH feature usages across time, per quarter. Overall, `require` always appears at least 1,000 times more than `revert` and `assert`. A sharp increase of the `require` usage (2,500 times) occurs from September 2018 to June 2019. The increase coincides with the release of `v0.4.22`, where Solidity provides developers with the ability to specify error messages in `require` (see Table 2). Furthermore, `revert` is more utilised than `assert` throughout the timeline. After the introduction of `try-catch` in `v0.6.0`, in 2019 (see Table 2), `try-catch` counterbalances the other two EH features, i.e., `revert` and `assert`. This happens from September 2019 to March 2020. In June 2020, the usage of `try-catch` surpasses the usage of `revert` and `assert`, respectively. The lead becomes more obvious in January 2021, when Solidity `v0.8.1` is released. A reason behind this may involve a newly introduced `try-catch` that provides developers with the ability to specify the reason for a failure using `Panic` and `Error` (see Table 2).

From Table 4 we observe that, on average, the use of `require` has the highest increase over time, followed by `assert` and `revert`. The most recent EH feature introduced in Solidity, i.e., `try-catch`, presents the lowest, yet increasing, growth rate over time. To check whether these differences are statistically significant, we use the Wilcoxon-test [33], since our data is not normally distributed. The results show that the resulting p-value is below the predetermined significance level (0.05), in the majority of our tests indicating significant differences between the pairs compared.

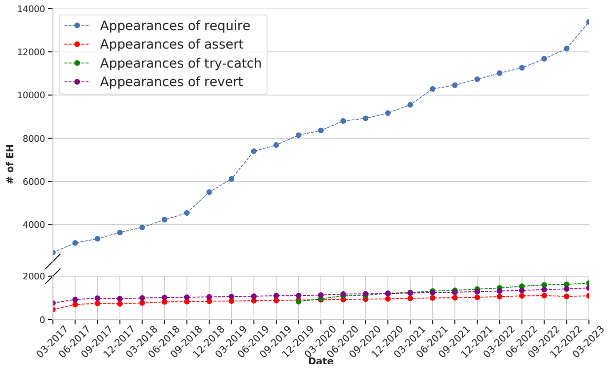

Figure 4: RQ3. Error-handling (EH) usage across years.

> **Answer to RQ3:** `require` has the highest increase across versions and years. From the second half of 2018, `require` has a sharp increase, which coincides with the release of `v0.4.22`, where developers can specify error messages in `require` and `revert`. With the introduction of `try-catch`, in `v0.6.0`, instantly, its usages become equal to those of `revert` and `assert`, surpassing them since June 2021.

4.4 RQ4: EH Misuses: Categories and Frequencies

Table 5 categorises EH misuses providing corresponding descriptions, the total cases where EH should have been used, and the number of misuses (i.e., missing EH) for each category. Note that our current approach *strictly* considers cases where *explicit* error handling is missing as *misuses*. We opt for this design option since this is the first empirical study that attempts to shed light on how much do developers consider Solidity's documentation, regarding the specification of the usages of EH features, and developers' willingness to use error handling; even in cases where implicit error handling (i.e., automatic transaction reversion) would be adequate e.g., in the case of an ECall or ECon misuse or error handling would be done by the SMTChecker (starting from compiler `v0.8.7`) regarding misuses relevant to `assert`. In the following paragraphs, we discuss in detail the misuse categories we identify.

External Call (ECall). When a smart contract performs a transaction, it usually makes calls to external contracts. According to the documentation, such calls should be be verified by developers using EH features such as `require` or `try-catch`, otherwise, potential problems may occur. For instance, if developers ignore checking whether an external call (e.g., `someAddress.call();`) can return a zero address, then the call may return some results that can modify the state of the caller contract in an unexpected manner. Examining our dataset, we found 110,655 external calls. For a significant number of external calls (63.63%), developers use neither `require` nor `try-catch`. Therefore, those situations represent potential failures that could happen due to an unchecked call return value [4]. Furthermore, consider that if an external call fails, it should be isolated (i.e., via EH), otherwise it could cause a DoS vulnerability [6].

Function Argument of `address` type (FAA)**.** To perform transactions, contracts use the `address` type to define the address of the sender and the recipient. According to the documentation, the `require` EH feature should be used to guarantee that the values of the `address` type are valid. For example, if the sender sends an amount to a receiver that has zero `address`, this amount will be sent to an account that does not exist and a loss of funds will occur. To ensure that the value of `address` is non-zero, developers need to use `require`, i.e., `require(recipient != address(0),"...");` In our dataset, we detect 31,018 coding situations where the function argument of `address` type should be verified by developers. For 89.43% of those situations, developers do not check if the address is not zero using `require`. Such cases can lead to attacks such as the the Nomad bridge attack [1] described in Section 2.

External Contract (ECon)**.** In Solidity, developers have the ability to create new smart contracts within other smart contracts, using the `new` keyword. Without using the `try-catch` EH feature, the creation of a contract within another contract can lead to software failures. Consider a smart contract A that uses a function to create a new smart contract B. If the deployment process is not done correctly and encounters an error, B may not be fully initialised, or might be left in an incomplete state, producing an exception. If `try-catch` is not used, the exception remains "uncaught", leading to an unexpected behaviour within the smart contract A. Overall, we identify 1,162 coding situations of external contracts in our dataset. From those situations, 80.37% do not involve `try-catch`, and can potentially cause software failures. Consider that there are also related situations that a vulnerability can happen due to an incorrect constructor name [9].

Array Allocation (AA). In Solidity, one can deploy a smart contract which allocates a memory array of user-supplied length. According to the documentation, if the length of a new array is not verified using `assert`, and the array's length is invalid, software failures may occur, including memory corruption [10]. From the 602 situations where array allocation takes place in our dataset, 66.67% do not involve `assert`. We observe that, although SMTChecker runs by default starting from Solidity `v0.8.7`, it cannot automatically detect such cases. Developers must explicitly configure the SMTChecker using command-line options, such as `-model-checker-targets` to guide the analysis.

Pop from Array (PA)**.** In Solidity, developers can delete elements from arrays using the `pop` function. However, if developers ignore using `assert` to test if the array is empty, the contract's execution will be stopped unexpectedly. Analysing our dataset, we found that in 181 coding situations where `pop` was used to delete elements from arrays, 90.62% do not use `assert`, as suggested by the documentation (for Solidity < `v0.8.7`).

Division by Zero (DZ)**.** For division by zero, developers should use `assert` according to the documentation (again for versions < `v0.8.7`). A division by zero can end the execution of the smart contract unexpectedly. In our dataset, we found 94 cases where division by zero may occur. However, 79.76% of them do not use `assert`. Such situations can be considered as potential exploits [11].

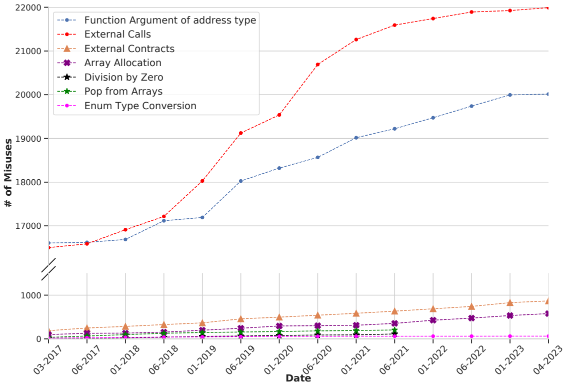

Figure 5: RQ5. Error-handling (EH) misuses per quarter.

`enum` **Type Conversion** (ETC)**.** According to the Solidity's documentation, developers should use `assert` when they try to convert a value that is too big or negative into an `enum` type. Consider a function that has a `uint` value and tries to convert this value into an `enum` value. If the conversion is unsuccessful, there is a possibility that this value will have a different type. To guarantee that the conversion will be successful, developers should use `assert`. In our dataset, from the 67 coding situations that include `enum` type conversions, 89.55% do not check the validity of the conversion. Despite the automatic execution of SMTChecker from Solidity `v0.8.7` and onwards, "`enum` type conversion" cases could not be detected automatically by the SMTChecker as in the case of AA.

Answer to RQ4: The category with the highest number of total cases that require error handling is the "external call" (110,655). However, for "external call" developers use error handling (63.63%) more than the other categories. The categories that developers mostly ignore error handling include "pop from array" (90.62%), "`enum` type conversion" (89.55%), and "function argument of `address` type" (89.43%).

4.5 RQ5: Evolution of EH Misuses In Practice

Figure 5, illustrates how the misuses identified in our dataset evolve over time, per quarter. In the first quarter of 2017, the number of "function argument of `address` type" (FAA) is 926 times higher than "external call" (ECall). In June 2017, ECall's frequency becomes higher than FAA's for the first time. From that point on, both have a steady increase over time with ECall leading the way. At the end (04-2023), the number of ECall is 31,277 higher than the FAA. The third most common misuse is "external contract" (ECon), being steadily ≈ 127% times higher than "array allocation" (AA). Through time, AA is on average ≈ 76% higher than "division by zero" (DZ), and ≈ 85% higher than "enum type conversion" (ETC). Note that `assert`-related misuses (DZ and "pop from array" – PA) frequencies are zeroed after June 2021. This happens due to the incorporation of the SMTChecker that runs by default in `solc` `v0.8.7`, and was released in June 2021. Recall that the SMTChecker automatically performs checks for DZ and PA, nullifying the use of `assert` for the corresponding coding situations.

Table 5: RQ4. Categories of error-handling (EH) misuses, corresponding descriptions, total cases where EH should have been used, and number of misuses for each category.

Missing EH Category (EH Misuse Category)	Category Description	Representative Misuse (Contract in Etherscan)	Misuse Description	Total Cases (#)	Missing EH (%)	EH Required
External Call (ECall)	Missing EH feature to validate a call to an external contract	`IRoutedSwapper`	Missing `require` or `try-catch` for external call `swapper.swapExactInput())`	110,655	70.417(63.63%)	`require` or `try-catch`
Function Argument of `address` type (FAA)	Missing `require` to check if an input of an `address` type is valid.	`StreetDawgs`	Missing `require` to check `from` in the code fragment `address from = address(uint160(_packedOwnershipOf(Id))`	31,018	27, 741(89.43%)	`require`
External Contract (ECon)	Missing `try` to test if the external contract is created correctly	`CapsuleFactory`	A new contract `Capsule capsuleCollection = new Capsule(name, symbol, tokenURIOwner, isCollection)` is created without using `try` to test the correct creation of the contract	1,162	934(80.37%)	`try-catch`
Array Allocation (AA)	Missing `assert` to test if the array allocated has a valid length	`VaultsRegistry`	Missing `assert` to test if `allVaults.length` in `address[] memory vaultsArray = new address[](allVaults.length);` is a valid number	602	401(66.67%)	`assert`
Pop from Array (PA)	Missing `assert` to test if `.pop()` is valid	`GenjiMonotagari`	The function `remove(Map storage map, address key)` that deletes the last key from `map.keys` does not have an assertion.	181	164(90.62%)	`assert`
Division by Zero (DZ)	Missing `assert` to check if the division is valid	`ZoomerRethPool`	In function `getRedeemValue(uint256 totalBurn)` there is not check if the `(balance / (5000 - totalBurn)` is `>0`	94	75(79.76%)	`assert`
`enum` Type Conversion (ETC)	Missing `assert` to validate if the conversion to the `enum` type is done correctly	`Hakiro`	In this contract, there is an `enum` type named `Step`. In function `setStep(uint step)`, there is a missing assertion to check if `sellingStep = Step(step);` is done correctly	67	60(89.55%)	`assert`

Table 6 presents descriptive statistics regarding the growth rates of EH misuses per quarter. FAA and ECall have the lowest median. This may seem slightly contradictory to the findings related to the corresponding misuses frequencies. However, the low growth rates involve the rise of using `require` over time to handle external calls and function arguments (see also Sec. 4.3). ECon has the next lowest median, meaning that developers relatively use `try-catch` to examine the creation of an external contract. DZ and PA have the highest median. This is because developers do not use `assert` to assess either if `pop` is valid or if a division by zero may take place. Note that in the case of DZ and PA, we examine the period of time prior to the inclusion of the SMTChecker. The next categories that have the highest median include `ETC` and `AA`. This indicates that developers should consider using `assert` more often to check array lengths and `enum` type conversions. Recall that SMTChecker have yet to automatically check the corresponding coding situations. Additionally, we performed the Wilcoxon-test [33] and found that the differences are statistically significant.

Table 6: RQ5. Error-handling (EH) misuse growth rates.

EH Misuse Category	Min	Max	Mean	Median	St. Dev.
External Call (ECall)	0.043	0.332	0.190	0.218	0.133
Function Argument of `address` type (FAA)	0.004	0.205	0.103	0.110	0.110
External Contract (ECon)	0.331	3.641	1.777	1.775	1.164
Array Allocation (AA)	0.306	4.897	2.081	2.045	1.590
Pop from Array (PA)	0.911	4.970	3.588	4.161	1.646
Division by Zero (DZ)	0.801	10.201	6.364	7.299	3.832
`enum` Type Conversion (ETC)	0.470	2.647	1.920	2.382	0.881

Answer to RQ5: Over time, the highest number of misuses involves the absence of EH for (1) validating calls to external contracts, (2) checking if the input of an `address` type is reliable. However, the corresponding growth rates are low, which indicates the increasing use of the EH features that can handle such cases. The impact of the introduction of the SMTChecker into the Solidity compiler is prominent here, with corresponding misuses' frequencies dropping ever since its introduction.

4.6 Key Take-aways and Suggestions

Amplifying EH usage in Solidity. Our results indicate that the usage of most EH features overtime is limited (except for `require`). Thus, we call for actions that can assist developers to engage in using those features, where required, avoiding potential EH misuses. Such actions may include the improvement of the Solidity documentation, by explaining, for instance, the usage of EH features by providing concrete examples, the development of ad-hoc tools that can automatically suggest EH usages where required, and the addition of extra checks into the SMTChecker.[6]

Stabilising Solidity's volatile nature. Our work highlights trends in the evolution of error handling in Solidity. Solidity designers change quite often EH features ma king Solidity volatile. Then, maybe it is difficult for developers to follow those changes, and ignore EH features. Thus, similar future studies to ours, as well as human surveys, could help Solidity designers to understand potential needs and, consequently, stabilise their framework.

Enhancing Solidity documentation. The EH misuses we discussed in Section 4.4 reveal that developers ignore or may find it difficult to understand the specification usage of EH features in the documentation. Therefore, there is a need to improve the documentation in terms of explainability, so that developers can understand the usage of EH features and remain engaged in EH usage (e.g., how to safely handle elements involved in transactions among smart contracts). Solidity designers could also add to the

---

[6] Note that SMTChecker already includes automatic checks replacing cases of `assert`.

documentation representative examples of real-world EH misuses and their impact in terms of reliability and security. We include examples in Section 4.4.

Introducing data-flow analysis. Incorporating data-flow analysis into `solc` could also help developers write more secure smart contracts. By employing data-flow analysis, developers will be able to to detect inputs coming from external calls, track their flow within the contract, and check if they reach any sensitive "sinks". Based on this information, they will be able to automatically identify where the use of `require` is essential. Furthermore, data-flow analysis can be performed by ad-hoc tools as done in other programming languages such as Java [21].

5 Threats to Validity

Construct validity. A potential threat to construct validity [54, 65] is posed by the use of the heuristic rules presented in Section 3.6. We mitigate this threat by validating the rules on a random sample of 80 smart contracts, following the approach described in Section 3.6. In each of the 80 cases, we observed correct matches between the AST nodes with EH features and the results from the heuristic rules. Our rules may miss potential misuses, but currently this is not something that one can measure in a candid way, i.e., there is no dataset-oracle to examine false negatives.

Another threat refers to potential false positives that our analysis may suffer from. To tackle this issue, two authors manually checked for false positives in the findings (e.g., whether a misuse identified by our heuristics is not an actual misuse). Specifically, for each misuse category, we randomly selected ten cases from our findings (70 in total). Then, the two authors examined each misuse to check for false positives. No false positives were identified.

Furthermore, we could not examine potential misuses of the `revert` feature since it is not straightforward for one to identify a misuse involving `revert`.[7] Therefore, we opted for excluding this feature from our analysis to reduce the risks of causing false positives/negatives.

The reader should also consider that, such as any study that involves manual analysis, our study may also suffer from human errors. To mitigate this threat the authors cross-checked their findings following the process explained in Section 3.4. Finally, our approach *strictly* categorises cases where error handling is not *explicitly* used by developers, according to Solidity's documentation. However, in practice, *implicit* error handling could be also sufficient. For instance, not every call to another contract has to be handled explicitly (e.g., consider an ECall or ECon misuse). We aim to enhance, in the future, our approach, so that we can also identify such cases, where implicit error handling is also adequate.

External validity. A threat to external validity of our study refers to the fact that our findings might not extend beyond the smart programs and Solidity versions investigated herein. However, to mitigate this threat we curated, to our knowledge, the largest publicly available benchmark of Solidity smart contracts to date. Moreover, we described in detail the methodology we used and made a replication package available to allow for future replication, reproduction and extension of our work to other smart contracts.

6 Related Work

Several studies investigate bugs and vulnerabilities of smart contracts (e.g., [22, 31, 42, 51, 53, 62, 67, 68]), and their automated detection and repair (e.g. [27, 35, 56]).

Other studies, focus on particular features of the Solidity programming language. For example, Chaliasos et al. [28] and Liao et al. [46] conduct an empirical study of 50M and 7.6M smart contracts, respectively, to explore the use of inline assembly. While, Liu et al. [47] conduct a large-scale study, on 3,866 smart contracts, to identify the use of transaction-reverting statements. Our study is broader though, focusing on different EH aspects, examining EH features' frequency and evolution.

Regarding error handling in Solidity, there are only a few studies available. Verheijke and Rocha [61] use a dataset of 26,799 smart contracts to examine the usage of Solidity functions such as `call`, `send`, and `transfer`. To evaluate whether those functions are used in a secure manner, the study also investigates the use of three Solidity guards, i.e., `assert`, `require`, and `revert`. As in our study, Verheijke and Rocha [61] also find that the developers of smart contracts mostly use `require`. Additionally, Wang et al. [63] analyse 172,645 real-world smart contracts to examine features of the Solidity programming language related to control flow, object-oriented programming, data structures and error handling. Contrary to the aforementioned studies, we conduct a thorough examination of the evolution of each Solidity EH feature and its usage on a dataset of 283K unique open-source smart contracts.

There are several studies that examine the EH mechanisms of other programming languages, including Java [36, 37, 43, 45, 49, 50, 57, 59, 64], C++ [23, 24, 66], Python [52], Ada [58], Swift [26], and Rust [38]. Most studies show that developers neglect the usage of error handling. In our study, we also find that developers use Solidity EH features rarely.

7 Conclusions

Our analysis of 283K smart contracts reveals that the usage of most EH features has been limited, although there is an upward trend in the usage of the `require` feature. The popularity of `require` indicates that programming language designers could consider the development of EH features that are more tailored to the purposes of each language. Furthermore, our analysis of the different versions of the Solidity documentation, as well as the analysis of the EH misuses found in practice, indicates that Solidity changes frequently, having a volatile nature. Additionally, smart contracts are particularly exposed to EH misuses, caused, for instance, from unchecked external calls. Based on our findings we identify four main areas of improvements: amplifying EH usage in Solidity, stabilising Solidity's volatile nature, enhancing Solidity documentation, and introducing data-flow analysis.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work has received funding from the European Union's

---

[7] The `revert` EH feature is mainly used to revert a transaction if a condition is not met. To do so, developers have to use `revert` in `if-else` conditionals, which are tailored to the smart contracts' logic.

Horizon Research and Innovation Programme under grant agreement No. 101070599 (SecOPERA project). Also, it is based upon work supported by the ERC Advanced Grant no.741278.

References

[1] n.d.. https://rekt.news/nomad-rekt/ Last access on 13/10/2023.

[2] n.d.. https://github.com/Solidity-ErrorHandling-Anonymous/solbench

[3] n.d.. https://docs.oracle.com/java/tutorial/essential/exceptions/throwing.html Last access on 12/10/2023.

[4] n.d.. https://swcregistry.io/docs/SWC-104 Last access on 13/10/2023.

[5] n.d.. https://swcregistry.io/docs/SWC-101 Last access on 15/10/2023.

[6] n.d.. https://swcregistry.io/docs/SWC-113 Last access on 15/10/2023.

[7] n.d.. . https://docs.soliditylang.org/en/v0.8.19/control-structures.html#error-handling-assert-require-revert-and-exceptions Last access on 07/10/2023.

[8] n.d.. https://github.com/Solidity-ErrorHandling-Anonymous/solbench/blob/main/src/ast_detector.py

[9] n.d.. https://swcregistry.io/docs/SWC-118 Last access on 15/10/2023.

[10] n.d.. https://soliditylang.org/blog/2020/04/06/memory-creation-overflow-bug/ Last access on 14/10/2023.

[11] n.d.. https://cwe.mitre.org/data/definitions/369.html Last access on 14/10/2023.

[12] n.d.. *Control Structures Solidity Documentation*. https://docs.soliditylang.org/en/v0.8.19/control-structures.html [Last access 15/10/2023].

[13] n.d.. *Daily Etherium transactions*. https://etherscan.io/chart/tx [Last access 31/10/2023].

[14] n.d.. *The Ethereum Platform*. https://ethereum.org/en/ [Last access 15/10/2023].

[15] n.d.. *Etherscan*. https://etherscan.io/ [Last access 31/10/2023].

[16] n.d.. *SMTChecker Documentation*. https://docs.soliditylang.org/en/v0.8.19/smtchecker.html [Last access 15/10/2023].

[17] n.d.. *Solidity*. https://docs.soliditylang.org/en/v0.8.0/ [Last access 15/10/2023].

[18] n.d.. *Version 0.4.13 Announcment*. https://blog.soliditylang.org/2017/07/06/solidity-0.4.13-release-announcement/ [Last access 16/10/2023].

[19] Manar Abdelhamid and Ghada Hassan. 2019. Blockchain and Smart Contracts. In *Proceedings of the 8th International Conference on Software and Information Engineering* (Cairo, Egypt) *(ICSIE '19)*. Association for Computing Machinery, New York, NY, USA, 91–95.

[20] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1170–1188.

[21] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 259–269. https://doi.org/10.1145/2594291.2594299

[22] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag, Berlin, Heidelberg, 164–186.

[23] Rodrigo Bonifácio, Fausto Carvalho, Guilherme N. Ramos, Uirá Kulesza, and Roberta Coelho. 2015. The use of C++ exception handling constructs: A comprehensive study. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 21–30.

[24] Kirsten Bradley and Michael Godfrey. 2019. A Study on the Effects of Exception Usage in Open-Source C++ Systems. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–11.

[25] Ryan Browne. 2021. *Ether, the world's second-biggest cryptocurrency, is closing in on an all-time high*. https://www.cnbc.com/2021/01/19/bitcoin-ethereum-eth-cryptocurrency-nears-all-time-high.html [Online; accessed 20-July-2023].

[26] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. 2018. How Swift Developers Handle Errors. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 292–302. https://doi.org/10.1145/3196398.3196428

[27] S. Chaliasos, M. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 705–717.

[28] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. A Study of Inline Assembly in Solidity Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 165 (oct 2022), 27 pages.

[29] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.* 53, 3, Article 67 (jun 2020), 43 pages.

[30] Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. 2022. Declarative Smart Contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 281–293.

[31] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345.

[32] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding Code Reuse in Smart Contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 470–479.

[33] William Jay Conover. 1999. *Practical nonparametric statistics* (3. ed ed.). Wiley, New York, NY [u.a.].

[34] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541.

[35] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM.

[36] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. 422–431. https://doi.org/10.1109/ICSE.2013.6606588

[37] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82–101.

[38] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

[39] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 204–215.

[40] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 3–16.

[41] Mingyuan Huang, Jiachi Chen, Zigui Jiang, and Zibin Zheng. 2024. Revealing Hidden Threats: An Empirical Study of Library Misuse in Smart Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 26, 12 pages. https://doi.org/10.1145/3597503.3623335

[42] Sungjae Hwang and Sukyoung Ryu. 2020. Gap between Theory and Practice: An Empirical Study of Security Patches in Solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 542–553.

[43] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. 2016. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Workshop on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 484–487.

[44] Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Measuring Ethereum Network Peers. In *Proceedings of the Internet Measurement Conference 2018* (Boston, MA, USA) *(IMC '18)*. Association for Computing Machinery, New York, NY, USA, 91–104.

[45] Joseph R. Kiniry. 2006. Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. In *Advanced Topics in Exception Handling Techniques*, Christophe Dony, Jørgen Lindskov Knudsen, Alexander Romanovsky, and Anand Tripathi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–300.

[46] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang, and Xiaosong Zhang. 2023. Large-Scale Empirical Study of Inline Assembly on 7.6 Million Ethereum Smart Contracts. *IEEE Transactions on Software Engineering* 49, 2 (2023), 777–801.

[47] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 630–641.

[48] Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K. Lahiri, and Isil Dillig. 2021. Demystifying Loops in Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 262–274.

[49] Cristina Marinescu. 2013. Should we beware the exceptions? an empirical study on the Eclipse project. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 250–257.

[50] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. 2016. Analysis of Exception Handling Patterns in Java Projects: An Empirical Study. In *Proceedings of the 13th International Workshop on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. 500–503.

[51] Gustavo A. Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. 2020. An Exploratory Study of Smart Contracts in the Ethereum Blockchain Platform. *Empirical Softw. Engg.* 25, 3 (may 2020), 1864–1904.

[52] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 24–35.

[53] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *USENIX Security Symposium*.

[54] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) *(EASE '18)*. Association for Computing Machinery, New York, NY, USA, 13–23.

[55] Alex Reinking and Ruzica Piskac. 2015. A Type-Directed Approach to Program Repair. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 511–517.

[56] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. 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* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 566–579.

[57] Martin P. Robillard and Gail C. Murphy. 2000. Designing Robust Java Programs with Exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications* (San Diego, California, USA) *(SIGSOFT '00/FSE-8)*. ACM, New York, NY, USA, 2–10.

[58] Alexander Romanovsky and Bo Sandén. 2001. Except for Exception Handling …. *Ada Lett.* XXI, 3 (sep 2001), 19–25. https://doi.org/10.1145/568671.568678

[59] H.B. Shah, C. Gorg, and M.J. Harrold. 2010. Understanding Exception Handling: Viewpoints of Novices and Experts. *IEEE Transactions on Software Engineering* 36, 2 (March 2010), 150–161.

[60] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API-Misuse Detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 265–275. https://doi.org/10.1109/MSR.2019.00053

[61] Darin Verheijke and Henrique Rocha. 2023. An Exploratory Study on Solidity Guards and Ether Exchange Constructs. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain* (Pittsburgh, Pennsylvania) *(WETSEB '22)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3528226.3528372

[62] Yilin Wang, Xiangping Chen, Yuan Huang, Hao-Nan Zhu, Jing Bian, and Zibin Zheng. 2023. An empirical study on real bug fixes from solidity smart contract projects. *Journal of Systems and Software* 204 (2023), 111787.

[63] Ziyan Wang, Xiangping Chen, Xiaocong Zhou, Yuan Huang, Zibin Zheng, and Jiajing Wu. 2021. An Empirical Study of Solidity Language Features. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 698–707. https://doi.org/10.1109/QRS-C55045.2021.00105

[64] Westley Weimer and George C. Necula. 2008. Exceptional Situations and Program Reliability. *ACM Transactions on Programming Language Systems* 30, 2, Article 8 (2008), 51 pages.

[65] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[66] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. 2023. Detecting Exception Handling Bugs in C++ Programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1084–1095.

[67] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 615–627.

[68] Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. 2023. *Identifying Solidity Smart Contract API Documentation Errors*. Association for Computing Machinery, New York, NY, USA.