2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678823

TRANSCODE: Detecting Status Code Mapping Errors in Large-Scale Systems

Wensheng Tang*, Yikun Hu*, Gang Fan*, Peisen Yao*, Rongxin Wu†,
Guangyuan Bai‡, Pengcheng Wang‡, and Charles Zhang*

*The Hong Kong University of Science and Technology
Hong Kong, China
{wtangae, yikunh, gfan, pyao, charlesz}@cse.ust.hk

† Xiamen University
China
wurongxin@xmu.edu.cn

‡Tencent, Inc.
China
{neobai, hadeswang}@tencent.com

***Abstract*—Status code mappings reveal state shifts of a program, mapping one status code to another. Due to careless programming or the lack of the system-wide knowledge of a whole program, developers can make incorrect mappings. Such errors are widely spread across modern software, some of which have even become critical vulnerabilities. Unfortunately, existing solutions merely focus on single status code values, while never considering the relationships, that is, mappings, among them. Therefore, it is imperative to propose an effective method to detect status code mapping errors.**

In this paper, we propose TRANSCODE to detect potential status code mapping errors. It firstly conducts value flow analysis to efficiently and precisely collect candidate status code values, that is, the integer values, which are checked by following conditional comparisons. Then, it aggregates the correlated status codes according to whether they are propagated with the same variable. Finally, TRANSCODE extracts mappings based on control dependencies and reports the mapping error if one status code is mapped to two others of the same kind. We have implemented TRANSCODE as a prototype system, and evaluated it with *5* real-world software projects, each of which possesses in the order of a million lines of code. The experimental results show that TRANSCODE is capable of handling large-scale systems in both a precise and efficient manner. Furthermore, it has discovered *59* new errors in the tested projects, among which *13* have been fixed by the community. We also deploy TRANSCODE in WeChat, a widely-used instant messaging service, and have succeeded in finding real mapping errors in the industrial settings.

I. INTRODUCTION

Many large-scale systems use status codes to represent program states. There are mappings among status codes since a system may describe the same program state at different levels of abstraction (e.g., disk failure v.s. IO error). These mappings are, unfortunately, error-prone. As an example, CVE-2010-0408[1] records a vulnerability of Apache Httpd v2.2 that is caused by a status code mapping error. In Figure 1, the variable *status* receives status codes indicating the result of handling a client request. In line 4, a different code HTTP_INTERNAL is returned to the caller (status code mapping) for any status codes that are not APR_SUCCESS. Such mappings make the server to keep re-processing incorrect requests[2]. As a consequence, a malicious client could launch a denial-of-service attack against a server. Worse still, these mapping errors are hard-to-be-detected since they seldom cause immediate symptoms (e.g., program crashes). The imperative of this research is to propose a practical solution to detect such status code mapping errors.

```
1 int ap_proxy_ajp_request(){
2   status = ap_get_brigade();
3   if (status != APR_SUCCESS){
4     return HTTP_INTERNAL;
5   }
6   ...
7 }
```

(a)

```
1 int ap_xml_parse_input(){
2   status = ap_get_brigade();
3   if (status != APR_SUCCESS) {
4     result = HTTP_BAD_REQUEST;
5     return result;
6   }
7 }
```

(b)

Fig. 1. (a) CVE-2010-0408: A status mapping error found in Apache Httpd v2.2 later causes a DoS vulnerability. (b) A correct error handling example that returns the correct status code. The status code alias names have been simplified.

To achieve this goal, we have to address two challenges. The first challenge is how to efficiently and precisely track the propagation status codes (**C1**). Simply tracking the propagation of status codes via data-flow analysis inevitably introduces false results because some infeasible paths disallow some status codes to pass through. However, employing path-sensitive analysis would worsen the scalability issue because, for every path, we need to check whether each status code can pass through by constraint solving. Owing to the presence of massive status codes in large-scale systems, it would be computationally expensive to solve feasibility queries for each path per status code. For instance, even just 34 status codes in the Linux kernel will touch all file systems and storage device drivers [1], burdening solvers with a high cost in feasibility validation.

* Yikun Hu and Peisen Yao are the corresponding authors.

[1]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0408

[2]https://tools.ietf.org/html/rfc7231#section-6.6

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00078

The second challenge is how to automatically infer the functionality-correlated status codes (**C2**). Mappings originate from program state shifting, represented by status codes, reflecting the correlation among them. Therefore, such correlation is essential knowledge for determining mapping errors. Unfortunately, such knowledge is not explicitly available in a program. In particular, as a convention, the developer defines functionality-correlated status codes (e.g., HTTP response codes) in various data structures, where many are undocumented and might even be distributed across random source files.

There are also numerous efforts on differentiating error status codes and non-error codes [1–7] that are not applicable for reasoning correlated status codes. In particular, some programs perform checks in the form of `if`$(err > 0)$ that coarsely take positive numbers as errors and vice versa. However, existing studies discard the correlation of status codes, e.g., HTTP response codes are only used to making server responses and consist of both errors and non-errors.

This paper proposes TRANSCODE that automatically detects status code mapping errors via inconsistencies in mappings. It firstly conducts a path-sensitive value flow analysis to collect candidate values of state codes. A domain inference algorithm is then applied according to whether one function transitively receives status codes from another. After that, TRANSCODE extracts mappings on control flow graphs and then reinforces them with domain information. At last, mapping errors are reported if one status code is inconsistently mapped to two other correlated relations.

Specifically, to tackle the challenge of the efficient and precise status code value collection (**C1**), we present an interpolation-guided algorithm [8, 9] that mitigates the costs of frequently invoking solvers. The insight of the approach is that some common infeasible paths could filter many invalid status codes. To tackle the challenge of reasoning correlations of status codes (**C2**), we propose an approach that automatically infers functionality-correlated status codes from value flows of status codes. We base it on a convention that each function outputs status codes of correlated functionalities only.

We evaluated TRANSCODE on five real-world software projects, each of which has 0.38-4.81 million lines of code. The experimental results show that TRANSCODE can precisely and efficiently obtain status code values during propagation and infer their functionality correlation. For each project, the above analysis is finished within half an hour. Moreover, it finds *59* new errors in the tested projects, *13* of which have been fixed by the developers.

In summary, the paper makes the following contributions:

- We introduce TRANSCODE, a novel approach for detecting mapping errors occurring among status codes. To the best of our knowledge, this work is the first research effort focusing on status code mappings.
- We implement the prototype of TRANSCODE, which is capable of handling real-world, large-scale applications precisely as well as efficiently.
- We conduct the experiments on five real-world software projects and discovered *59* new errors, among which *13* have been fixed by the community.

II. TRANSCODE IN A NUTSHELL

In this section, we first use an example to illustrate the incorrect status code mapping problem. We then present an overview of our approach to resolving this problem.

Figure 2(a) is a code snippet that contains a mapping error. Lines 1-8 are the status codes defined via two types of data structures. In function f_2, two mappings are present, giving two branches that go to different return statements. In particular, upon a status code `DECLINED` returned by function f_0, a new status code `HTTP_BAD_REQUEST` is returned, composing a mapping from `DECLINED` to `HTTP_BAD_REQUEST` on Lines 18-19. Similarly, a mapping from `DECLINED` to `HTTP_OK` can be found on Lines 20-21. However, these two mappings de facto make an error because the same status code `OK` is mapped to two different HTTP response codes. Hence, mapping errors like this and the one in Section I not only lead to error mishandling but also compromise the program.

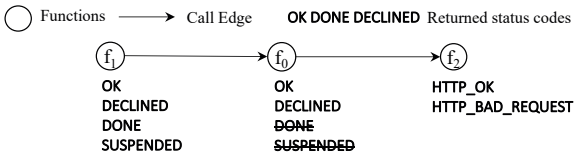
How to detect the above mapping error is a non-trivial task. It first requires precisely revealing the propagation of status codes. For instance, the value check at Line 13 decides whether the variable *rv* will continue to propagate some status code values. A second imperative is to determine which status codes represent the same sort of functionality because the inconsistencies only occur when two destination status codes are functionality-correlated. For example, `HTTP_OK` and `HTTP_BAD_REQUEST` are highly-correlated since they are all HTTP response codes. The challenging part of this goal is how to automatically infer such correlations without any prior knowledge because they are mostly project-specific heuristics. These heuristics for grouping status codes include naming, data structures, and values that often vary widely from one code base to another. Needless to say that providing the knowledge requires expertise and tedious examination of the source code.

To conquer the beasts, we propose a systematic approach, namely TRANSCODE, of which a workflow is given in Figure 3. We begin with collecting the status code returned each function as the initial summaries via a path-sensitive value-flow analysis (§IV-A). These summaries are further examined for inferring the underlying correlations of status codes, that is, whether two status codes are of the same kind (§IV-B). At last, aided with the correlations, we can demystify mapping errors by exposing the inconsistencies between status codes mappings where destination status codes are correlated (§IV-C).

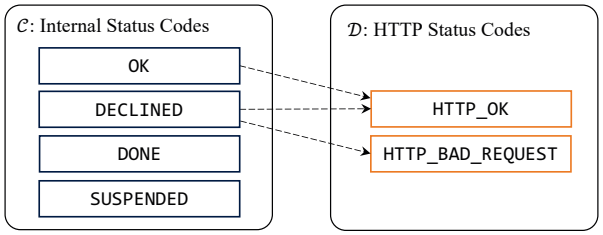
As a running example, Figure 2(b) exemplifies how our value-flow analysis tracks the propagation of status codes. Specifically, function f_0 inherits the status codes from function f_1, thus resulting in a summary $\{$`OK`, `DECLINED`, `DONE`, `SUSPENDED`$\}$. The summary is then reduced to $\{$`OK`, `DECLINED`$\}$, because neither `DONE` nor `SUSPENDED` can become a return value, as they are obviously filtered by the condition $rv >$ `DONE`. To further

```
1.  #define OK 0
2.  #define DECLINED -1
3.  #define DONE -2
4.  #define SUSPENDED -3
5.  enum HTTPStatusCodes {
6.      HTTP_OK = 200,
7.      HTTP_BAD_REQUEST = 403
8.  };
9.  /* f1 returns { OK, DONE, DECLINED, SUSPENDED} */
10. int dialup_send_pulse(); // f1
11. int dialup_handler() { // f0
12.     rv = dialup_send_pulse(); // f1
13.     if (rv > DONE) {
14.         return rv;
15.     }
16. }
17. int hook() { // f2
18.     if (dialup_handler() != OK)
19.         return HTTP_BAD_REQUEST;
20.     if (dialup_handler() != SUSPENDED)
21.         return HTTP_OK;
22. }
```

(a) Simplified code example from *mod_dialup.c* in Httpd.



(b) Results of value-flow analysis



(c) Status code mappings

Fig. 2. A motivation example.

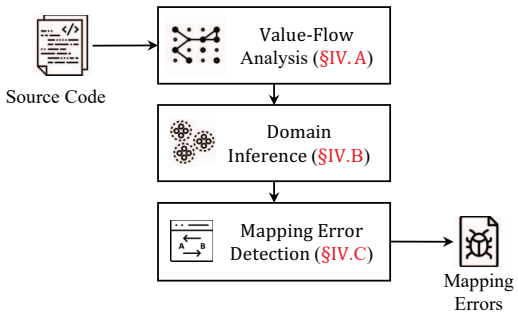


Fig. 3. System Overview of TRANSCODE

group correlated status codes, we unify the overlapping sets of status codes returned by some functions, such that two groups of status codes $\{\texttt{OK}, \texttt{DECLINED}, \texttt{DONE}, \texttt{SUSPENDED}\}$ and $\{\texttt{HTTP\_OK}, \texttt{HTTP\_BAD\_REQUEST}\}$ are established and clearly distinguishable from one another. As revealed by control-flow paths at Lines 18-21, two status code mappings are identified as well (Figure 2(c)). Thus, the two mappings compose an inconsistency, since they both occur between two groups of status codes. That is, the inconsistency of two mappings, $\texttt{DECLINED} \mapsto \texttt{HTTP\_OK}$ and $\texttt{DECLINED} \mapsto \texttt{HTTP\_BAD\_REQUEST}$, reveals the culprit of a mapping error.

III. PROBLEM FORMULATION

In this section, we first present the fundamental definitions and terminologies, and then formulate the mapping error detection problem.

A. Preliminaries

We begin with the formal definitions of status codes and their domains:

Definition III.1 (Status Code)**.** A status code c is a constant integer $c \in \mathbb{Z}$ served as program state indicators.

Definition III.2 (Status Code Domain)**.** A status codes domain is a set of status codes $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$, describing the states of a functionality-independent software feature[3]. For simplicity, we abbreviate "status code domain" to "domain" for short in the rest of the paper.

Example III.1. Consider the program in Figure 2. $\texttt{HTTP\_BAD\_REQUEST}$ is a status code from HTTP domain $\mathcal{D}$, representing a type of HTTP response errors. The domain is also used to differentiate status codes with the same value but different meanings.

A status code mapping occurs when a status code stops propagating, and in the same function, a new status code starts to propagate. Formally, a mapping instance can be defined in terms of status codes and domains:

Definition III.3 (Status Code Mapping)**.** The mapping relation is formalized as a *partial function* [11] in mathematics between a domain $\mathcal{C}$ and a co-domain $\mathcal{D}$, denoted by $\Gamma : C \rightharpoonup D$, such that there exists at least one status code $c \in \mathcal{C}$ and another status code $d \in \mathcal{D}$ satisfying $\Gamma(c) = d$. In the remaining sections, we use an alternative representation to explicitly mark a mapping with its corresponding domain and co-domain: $(c, \mathcal{C}) \mapsto (d, \mathcal{D})$[4][5].

Example III.2. As exemplified in Figure 2, there is a mapping $(\texttt{DECLINED}, \mathcal{C}) \mapsto (\texttt{HTTP\_BAD\_REQUEST}, \mathcal{D})$ at Lines 18-19, where $\texttt{DECLINED}$ is a status code received from a call site to f_1 at Line 18.

B. Problem Statement

Mapping errors originate from inconsistencies. As illustrated in the former section, one status code might be mapped

[3]It is a formalized definition originated from Apple [10].

[4]The mapping of a mapping might be inferred across many functions but not specific to local facts within one function.

[5]The pairing representation always references a unique status code in a program, regardless of its value equivalence over other status code values.

into two different status codes of the same domain. In this paper, we aim to automatically detect such inconsistencies. Formally, we define the mapping error as follows.

Definition III.4 (Mapping Error)**.** A mapping error occurs when *multivalued* mappings are detected, making the mapping relation no longer a partial function. Formally, there exist $(c, \mathcal{C}) \mapsto (d, \mathcal{D})$, and $(c, \mathcal{C}) \mapsto (d', \mathcal{D})$, where $d \neq d'$.

Example III.3. At Lines 17-20 in Figure 2, the two inconsistent mappings are $(\texttt{DECLINED}, \mathcal{C}) \mapsto (\texttt{HTTP\_OK}, \mathcal{D})$ and $(\texttt{DECLINED}, \mathcal{C}) \mapsto (\texttt{HTTP\_BAD\_REQUEST}, \mathcal{D})$, because they are across two consistent domains $\mathcal{C}$ and $\mathcal{D}$, but the destination status codes diverge.

Detecting mapping errors raises two major challenging issues. The first is how to efficiently and precisely track the propagation of status codes. To validate each propagation, we have to consider its path-feasibility. Assuming a path condition is ϕ and a return variable is rv, there would be an excessive amount of path-feasibility queries subject to each status code value of the form $\phi \wedge (rv = c_1), \ldots, \phi \wedge (rv = c_n)$. Compared to solving the path condition ϕ alone, those massive queries for each status code can overload the solvers, leading to a significant scalability problem.

The second issue in detecting mapping errors is automatically inferring the *domain* $\mathcal{C}$ of status codes. Providing that status code definitions are capricious in large software systems, a simple pattern-based code scan will bring false positives or false negatives and requires expertise to adapt to broader software systems.

Based on the discussion above, our problem in terms of mapping error detection can be summarized as follows:

> Given a program P, efficiently and precisely track the propagation of status codes and infer the underlying domains of each status code.

IV. Approach

This section presents a value flow analysis for status code collection, domain inference from collected status codes, and the mapping error detection process.

A. Value-Flow Analysis For Status Code Collection

Path-sensitive Value Flow Analysis. The first stage of TRANSCODE is a value flow analysis for analyzing the status code values returned by each function, which is the key input of domain inference (§IV-B). Specifically, TRANSCODE selectively searches from the definition of constant integer values to the return statements that they flow to. Similar to previous program analysis [12, 13], our design adopts bottom-up summary-based techniques to boost the performance of the inter-procedural analysis.

At a high level, the value flow analysis summarizes intra-procedural sub-paths to build the full paths from sources to sinks, where a source is the origin of a status code and a sink is the end of its propagation. More concretely, our analysis tracks and composes the following sub-paths, where $\rightsquigarrow$ operator denotes if some value v_1 can reach to another value v_2 via assignments.

- $c \rightsquigarrow$ **return** v: A path from a constant integer c to return value.
- $c \rightsquigarrow$ **call** $f(\ldots, arg, \ldots)$: A path from a constant integer c to a callee's argument arg.
- $arg \rightsquigarrow$ **return** v: A path from a function argument to return value.
- $v_1 \leftarrow$ **call** $f(\ldots) \rightsquigarrow$ **return** v_2: A path from a callee's return values to a return value of the current function.

During the analysis, if a composite path of the above sub-paths is $c \rightsquigarrow$ **return** v, the status code c is added to the summary of all status codes returned by function f, denoted as $Ret(f)$. Moreover, by leveraging the function summary, a caller function can inherit the callee's results if there is a path $v_1 \leftarrow$ **call** $g(\ldots) \rightsquigarrow$ **return** v_2. Formally, two inference rules collecting status codes of function f are given:

$$\frac{\ell_1 : c \rightsquigarrow \ell_2 : \textbf{return } v}{c \in Ret(f)} \tag{1}$$

$$\frac{\ell_1 : v_1 \leftarrow \textbf{call } g(\ldots) \rightsquigarrow \ell_2 : \textbf{return } v_2}{Ret(g) \subseteq Ret(f)} \tag{2}$$

To obtain a more precise result from the above rules, we can filter infeasible propagation via a path-sensitive analysis. We follow the previous work on a compositional and efficient encoding to collect each return statement's path condition [12]. Specifically, assume the path condition is ϕ and there exists a set of candidate status codes $c_1, c_2, \ldots, c_n$ to be verified. Then, we construct and solve feasibility queries in the form of $\phi \wedge (rv = c_1), \phi \wedge (rv = c_2), \ldots, \phi \wedge (rv = c_n)$, respectively. If one query $\phi \wedge rv = c_i$ is UNSAT, then we can exclude c_i from $Ret(f)$.

Example IV.1. Consider the function f_0 in Figure 2. After running a value flow analysis, we can acquire the initial summary of f_0: $Ret(f_0) = \{\texttt{OK}, \texttt{DECLINED}, \texttt{DONE}, \texttt{SUSPENDED}\}$. We then construct path-feasibility queries for each status code and solve $\phi \wedge (rv = \texttt{OK})$, $\phi \wedge (rv = \texttt{DECLINED})$, $\phi \wedge (rv = \texttt{DONE})$, and $\phi \wedge (rv = \texttt{SUSPENDED})$ separately. The finalized summary of f_0 is:

$$Ret(f_0) = \{\texttt{OK}, \texttt{DECLINED}\}.$$

Interpolation-guided Refinement. Recall that to compute the summary for a function f, we need to solve instances of path-feasibility queries of the form $\phi \wedge (rv = c_i)$. However, if the number of functions and the number of status codes are both large, the path-sensitive value flow analysis can result in a huge number of SMT solver calls, leading to performance issues.

To potentially reduce the number of solver calls, we present an interpolation-based optimization [8, 9]. The key insight of our optimization is that the facts disallowing some status code values to be propagated can be used to determine the feasibility of other status codes, thus enabling us to potentially prune unnecessary calls to solvers.

Algorithm 1: Interpolation-guided Refinement

Input: The path condition ϕ and a set of constraints $S = \{rv = c_1, \ldots, rv = c_n\}$
Output: Decide the satisfiability of each $\phi \wedge rv = c_i$ $(1 \leq i \leq n)$

```
1  foreach ψ_i ∈ S do
2      if φ ∧ ψ_i is UNSAT then
3          I ← GetInterpolant(φ, ψ_i);
4          foreach ψ_j : (rv = c_j) ∈ S do
5              // no solver required
6              if I[c_j/rv] = false then
7                  eliminate ψ_j from S;
8      else
9          // c_i will be added to Ret(f)
10         Mark φ ∧ ψ_i as satisfiable;
```

Definition IV.1. An interpolant for a pair of inconsistent formulas (A, B) (i.e., $A \wedge B = \mathsf{UNSAT}$) is a formula I satisfying (1) $A \Rightarrow I$ (2) $I \wedge B = \mathsf{UNSAT}$ (3) $Vars(I) = Vars(A) \cap Vars(B)$, where $Vars(P)$ denotes the variables referenced by predicate P.

Note that, for a pair of inconsistent first-order formulas A and B, there must exist at least one interpolant [14]. For example, $a < 1 \wedge a = 4$ is UNSAT, and an interpolant for $(a < 1, a = 4)$ can be $a < 3$.

In our context, the constraints are of the form $\phi \wedge rv = c_i$, where ϕ is the path condition and c_i is a constant status code. Consequently, if $\phi \wedge rv = c_i$ is UNSAT, an interpolant I of $(\phi, rv = c_i)$ must exist, and it only references rv, i.e., $Vars(I) = \{rv\}$.

Thus, the key idea underlying our approach is to use the interpolant I of $(\phi, rv = c_i)$ to potentially prune the satisfiability queries of other constraints $\phi \wedge rv = c_j$ $(i \neq j)$. In particular, if $I \wedge rv = c_j$ is UNSAT, then $\phi \wedge rv = c_j$ must also be UNSAT, because $\phi \Rightarrow I$. Notably, since I references only one variable rv, it is unnecessary to invoke a solver to determine the satisfiability of $I[c_j/rv]$.

Based on the above idea, we apply Algorithm 1 to optimize the computation of summary $Ret(f)$. For instance, assume an interpolant $rv < c_i$ is available upon an unsatisfiable formula $\phi \wedge rv = c_i$, which provides a detailed explanation of why the predicate $rv = c_i$ conflicts with the path condition (Line 3). Other predicates might potentially conflict with the interpolant as well, indicating their unsatisfiability of not invoking a solver (Lines 4-7). Note that an interpolant might not always be the most precise. Still, since the interpolant involves the return variable *rv* only, it would be showing great potential to be precise and conflict with other queried formulas.

Example IV.2. When checking the path-feasibility of status code `DONE`, the solver answers that $\phi \wedge (rv = \texttt{DONE})$ is UNSAT At the same time, an interpolant $I = (rv \leq \texttt{DONE})$ recorded during solving serves as a precondition subject to rv. This interpolant can then be applied to filter other infeasible status code predicates, for example, $(rv = \texttt{SUSPENDED})$. A conflict between the interpolant and the predicate is observed in $I[\texttt{SUSPENDED}/rv]$. Therefore, we conclude that `SUSPENDED` is pruned without a solver: $Ret(f_0) = \{\texttt{OK}, \texttt{DECLINED}\}$.

Remarks We also record status code naming from macros, enumerable types, and various data structures in the preprocessing stage at the source code level and feed it to the post-processing stage (value flow analysis). Our analysis, therefore, can distinguish all status codes even with the same value but different names.

B. Status Code Domain Inference

From §IV-A, the returned status codes of each function is collected. However, it remains to distinguish whether some status codes belong to the same domain for mapping error detection.

Domain-specific expertise may help define these domains manually, yet it requires considerable effort and may cause false results. The key reason is the existence of a massive amount of status codes, and its declaration method varies from projects favoring different programming conventions. For instance, multiple domains of status codes are declared in the header file *httpd.h*, and there are also status codes defined in multiple files that compose the same domain. Besides, these domains can be declared in various data structures such as macros, global variables, enumerable types, and class members, increasing the difficulty for developers to provide such specifications.

Return Conventions of Status Codes. To automatically infer the domains from the programs, we utilize an observation about the return convention of status codes. Specifically, as found in past research [1, 4], some implementation-specific status codes are internally used at callee functions and will not continue to propagate to the caller functions. In order words, a function only returns correlated status codes within the same domain.

Example IV.3. Figure 4 presents the propagation of status codes for Figure 2. Function f_0 takes status codes from f_1 and continues to propagate them. Therefore, both functions f_0 and f_1 return status codes within the same domain.

Unification-based Domain Inference. By the observation, we can infer each function's domain by a unification-based Algorithm 2. The key idea is that we unify all non-disjoint sets of returned status codes for each function and obtain the resulting disjoint sets as *domains*.

Example IV.4. Consider the domain inference of functions f_0, f_1, f_2. From §IV-A, we obtained the sets of status codes for each function $Ret(f_i)$. By unifying all non-disjoint

Algorithm 2: Domain Inference

Input: Returned status codes of each function $Ret(f_i)$
Output: Sets of status code domains $\mathbb{D}$

1 $\mathbb{D} = \{\mathcal{A} = Ret(f_0), \mathcal{B} = Ret(f_1), \ldots, \mathcal{N} = Ret(f_n)\}$;
2 **foreach** $(\mathcal{I}, \mathcal{J}) \in \mathbb{D} \times \mathbb{D}, \mathcal{I} \neq \mathcal{J}$ **do**
3 // pairwise set intersection
4 **if** $\mathcal{I} \cap \mathcal{J} \neq \varnothing$ **then**
5 $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{J}$;
6 eleminate $\mathcal{J}$ from $\mathbb{D}$;
7 **return** $\mathbb{D}$;

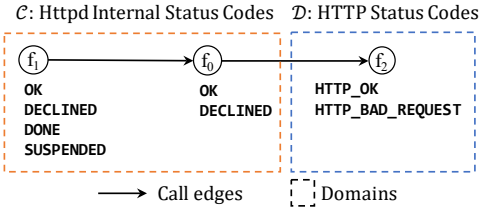


Fig. 4. Propagation of status codes in Figure 2. Status codes of functions f_1, f_0 have no overlap of status codes of f_2.

sets, we obtain domain facts as follows:

$$\mathcal{C} = \{\texttt{OK}, \texttt{DONE}, \texttt{DECLINED}, \texttt{SUSPENDED}\}$$
$$\mathcal{D} = \{\texttt{HTTP\_OK}, \texttt{HTTP\_BAD\_REQUEST}\}.$$

The algorithm above mainly utilizes set intersections and unions. However, previous program analyses based on set operations have been proved with high time and space complexity [15–17], giving the massive functions and status codes in large-scale software systems. In consequence, pairwise set-intersections among all sets of returned status codes might lead to deficiencies.

Value-Flow-Aided Domain Inference. As exemplified in the previous example, the domains of different status codes can be identified by performing pair-wise set unions of the collected status codes. However, a key problem with Algorithm 2 is that some set intersections at Line 4 are redundant if the algorithm is aware of the propagation facts during the value flow analysis. In particular, we can directly find out if the status codes returned by some functions overlap or not using intermediate value flow information. For example in Figure 4, function f_0 takes status codes from f_1 so that status codes in $Ret(f_0)$ and $Ret(f_1)$ are in the same domain, without the need to examine their concrete status codes. With this observation, we propose to optimize domain inference by recording summaries during value-flow analysis.

To leverage the observation, we further identify two subscenarios and make the corresponding reactions to accelerate domain inference. The three types of intermediate value flow information are illustrated in Figure 4:

- *Overlapping*. During value flow analysis, it is found that f_0 returns status codes directly received from f_1, indicating $Ret(f_0) \setminus Ret(f_1) \neq \varnothing$. For this case, status codes from both functions must be in the same domain.
- *Separation*. During value flow analysis, it is known f_2 receives status codes from f_0 but does not continue to return them, i.e., $Ret(f_0) \cap Ret(f_2) = \varnothing$. Therefore, there is no need to perform set intersections between $Ret(f_0)$ and $Ret(f_2)$.

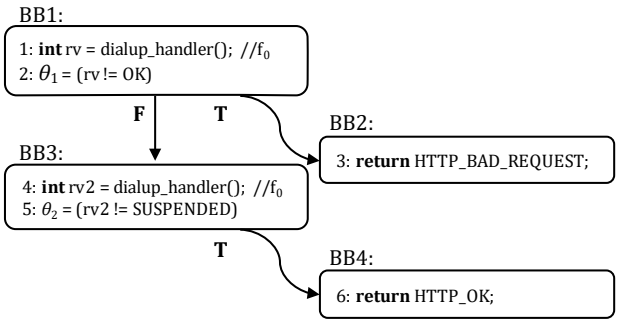


Fig. 5. The control-flow graph of function f_2 in Figure 2

The above relations can be directly recorded during value flow analysis without additional cost. Before Line 2, we can query the value flow summary if two sets are in the above relations. Once an *overlapping* relation is detected, we can skip the set intersections at Line 4 and directly merge all included status codes as one domain. Similarly, if *separation* relation is found, i.e., two sets belonging to two distinct domains, there is no need to test the two sets, thus skipping entry to the loop and Line 2. Therefore, our optimization saves on the costs of all-pairs square-time set intersections.

Example IV.5. Figure 4 has present value flow summaries composed of f_0, f_1, f_2 and their status codes returned. Owing to the overlapping relation between f_0 and f_1, we only need to take the union of two sets $Ret(f_0) \cup Ret(f_1)$ as a domain. Besides, there exists separation relation between f_0 and f_2 because f_2 no longer propagates f_0's status codes. We might skip the comparison of f_0 and f_2 entirely. Compared to $C_3^2 = 3$ pairwise set intersections in the original algorithm, we make no set intersections at all with the help of value flow summaries.

C. Mapping Error Detection

Once the domain information is exposed to the detector, the mapping error detection problem can be reduced to discovering domain inconsistencies among code-to-code mappings.

Mapping Identification. As also mentioned in IV-B, mappings occur in a function when status codes do not continue to propagate, and some new status codes emerge. For example, in Figure 4, function f_0 is a callee of f_2 but $Ret(f_0)$ and $Ret(f_2)$ belong to different domains. Since there is no common status code between $Ret(f_0)$ and $Ret(f_2)$, it is possible that mappings from status codes in $Ret(f_0)$ to those in $Ret(f_2)$ exist. The next question is how to infer concrete mappings in function f_2.

Algorithm 3: Mapping Identification in a Function

Input: A statement ℓ_{rcv} that receives status codes R from a domain $\mathcal{C}$
Input: A return statement ℓ_{ret} that outputs a status code d in another domain $\mathcal{D}$
Output: Mapping relation Γ

```
1 if ℓ_rcv can reach ℓ_ret on the control-flow graph then
2     Collect path conditions Φ_{ℓ_rcv → ℓ_ret};
3     foreach c ∈ R do
4         if Φ_{ℓ_rcv → ℓ_ret} ∧ rv = c is SAT then
5             Γ ← {c ↦ d};
6 return Γ;
```

We propose to identify these mappings from a control flow graph of one function via Algorithm 3. In Figure 5, at each statement that receives status codes from f_0 that are from one domain, e.g., Line 1 of Figure 5, we can traverse the control-flow graph to check if that statement can reach some statements that returns status codes of another domain. For instance, our analysis begins searching from Line 1, where the return value rv can be `OK` or `DECLINED`. Along the control-flow path, a return statement at Line 3 does return a status code of the HTTP domain. It can be further inferred that reachability is guarded by $\neg(rv \neq \texttt{OK})$. The guard is satisfiable when filling $rv = \texttt{DECLINED}$, where `DECLINED` is a status code from f_0. Therefore, a mapping `DECLINED` $\mapsto$ `HTTP_BAD_REQUEST` is found. Similarly, another mapping `DECLINED` $\mapsto$ `HTTP_OK` is constructed in a path from Line 5 to Line 6. Note that our identification is language-agnostic because it only as Algorithm 3 only takes a control-flow graph and mapping-related statements as input.

Inconsistency Detection. In this procedure, the goal of inconsistency detection is to detect whether destination status codes diverge. Hence, once all mappings are collected, we only need to compare mappings within the same source and destination domain. For every mapping originating from the same status code that maps to two other different status codes of the destination domain, TRANSCODE reports it as an error:

$$\frac{(c_i, \mathcal{C}) \mapsto (c_j, \mathcal{D}) \qquad (c_i, \mathcal{C}) \mapsto (c_k, \mathcal{D}) \qquad c_j \neq c_k}{\{c_i \mapsto c_j, c_i \mapsto c_k\} \in \mathcal{E}} \quad (3)$$

Example IV.6. Considering the example in Figure 2, we extract two mappings from the two functions, respectively:

$$(\texttt{DECLINED}, \mathcal{C}) \mapsto (\texttt{HTTP\_OK}, \mathcal{D})$$
$$(\texttt{DECLINED}, \mathcal{C}) \mapsto (\texttt{HTTP\_BAD\_REQUEST}, \mathcal{D})$$

We infer the belonging domain of the three status codes, and the two mappings are indeed from the same status code to two destination status codes of the same domain. Hence, we conclude that there is some inconsistency. The inconsistency also matches our manual inspection that status codes of a request filter are inconsistently mapped to two different HTTP response codes.

TABLE I
Code metrics of evaluated projects

Project	Domain	Version ID	Language(s)	KLoC
WolfSSL	Encryption	44e575b	C	926
Httpd	Web Server	abe9502	C	382
SQLite	Database	a90d817	C	383
GRPC	RPC Library	81299e2	C, C++	1,967
Asterisk	Voice over IP	2a6a280	C, C++	4,812

V. IMPLEMENTATION

We have implemented TRANSCODE on top of the LLVM 3.6 framework [18]. the Z3 SMT solver [19] with extensive use of Standard Template Library (STL).

Pointers, Loops and *errno* In the value flow analysis (§IV-A), there may be status codes stored to or loaded from pointer variables. As presented in previous approaches on value-flow analysis [12, 20, 21], TRANSCODE adopts a fast unification-based alias analysis [22] to resolve C style function pointers and employs a class hierarchy analysis [23] to resolve virtual function calls. For loops, we unroll each loop once in the control flow graphs and treat the remaining iterations guarded by the loop guard. This tradeoff is sound for status codes collection and mapping identification because our analysis permits propagation of each feasible status code only through constant's propagation. Besides, we also consider the semantics of system-wide status code *errno* in the implementation. However, since *errno* can also be set by library functions, static results could be incomplete. Nevertheless, as a common practice [12, 24], our analysis does treat *errno* as a global variable by copying it as an auxiliary variable to complement its value flow.

Pre-processing Status Code Values. Most status codes do not have corresponding variable names since they are usually defined using macros and various data structures. Specifically, in LLVM IR, all status code identifiers have already been optimized as unnamed integers. Therefore, in addition, the Clang [25] compiler frontend is employed to preprocess the source code to obtain names of status codes, and thus is able to differentiate among status codes with identical integer values.

VI. EVALUATION

In this section, we evaluate TRANSCODE by answering the following questions:

RQ1: How efficiently does TRANSCODE analyze on real-world programs?
RQ2: How precisely does TRANSCODE infer domains of status codes?
RQ3: Can TRANSCODE detect real-world status code mapping errors?

A. Experiment Setup

To answer the three research questions, We conduct three experiments for the above three research questions on real-world projects. For the experiment subjects, we choose five representatives, widely-used and large-sized projects from

TABLE II
Analysis Performance (in Seconds). VFA_S and VFA_I denotes value flow analysis using pure solver or applying interpolants-guided refinement, respectively. DI denotes the stage of domain inference. TD denotes the stage of mapping error detection. The last two columns are the total running times without and with refined value flow analysis.

Project	VFA_S	VFA_I	DI	TD	$Total_S$	$Total_I$
WolfSSL	4,465	450	3	764	5,232	1,217
Httpd	454	285	2	27	483	314
SQLite	329	83	4	24	357	111
GRPC	880	791	15	5	900	811
Asterisk	1,169	805	248	92	1,509	1,145
Total	7,297	2,414	272	912	8,481	3,598

different domains: WolfSSL, Apache Httpd, Google RPC, Asterisk and SQLite, ranging in size from 0.38 MLoC to 4.81 MLoC. Table I shows more details of these subjects. All projects are the latest version from its code repositories at the time of the experiment.

All experiments are conducted on a computer with an Intel Quad-Core i5-6500 3.20GHz CPU and 8 GB RAM.

B. Performance

We divide our analysis into three stages: (i) extracting status code values for each function by value-flow analysis; (ii) inferring domains; and (iii) mapping error detection.

Table II shows the analysis time of each stage (VFA, DI and TD), and the overall analysis time with two different value flow analyses ($\textbf{Total}_\textbf{S}$, $\textbf{Total}_\textbf{I}$). Overall, the TRANSCODE approach ($\textbf{Total}_\textbf{I}$) finishes analyzing each project in less than twenty-one minutes on a moderate computer, which demonstrates that our approach scales well for real-world projects. The time consumption is mostly dominated by the value-flow analysis (VFA, §IV-A) stage (2,414/3,598=67%). This is reasonable since the value-flow analysis is a context- and path-sensitive inter-procedural analysis, and it relies on a heavy-weight SMT solver for resolving path feasibility. Basically, the analysis time for a project is related to its size and the number of status codes it has. For example, the analysis for WolfSSL, which has the largest number of status codes and functions, is the most time-consuming of the five projects.

By comparing the two different value-flow analyses ($\textbf{VFA}_\textbf{S}$ and $\textbf{VFA}_\textbf{I}$), it is obvious that the interpolants-guided filtering (§IV-A) is effective in reducing the cost. For example, the interpolation approach is able to reduce 77% (=1-1,217/5,232) of the analysis cost for WolfSSL. This is because the approach can speed up the path feasibility verification process by leveraging the interpolant of UNSAT queries to skip other similar UNSAT queries, thus improving the efficiency.

The time cost for domain inference (DI, §IV-B) and mapping error detection is considerably small. For domain inference, our algorithm, relying on set operations, has a square time complexity in the worst cases. However, in our experiment, the time cost is small (2-15s) for four out of five projects. It only becomes non-trivial (248s) for Asterisk, which has the largest number of domains.

TABLE III
Inferred facts for each project. **Status Codes** denote the total inferred status codes. **Domains** denote the number of inferred domains.

Project	Functions	Status Codes	Domains
WolfSSL	13,889	79,607	24
Httpd	9,302	9,240	17
SQLite	3,301	3,485	78
GRPC	19,734	2,653	14
Asterisk	13,704	13,996	95

Time consumption of mapping error detection (TD, §IV-C) typically remains at a low level, because our algorithm essentially takes the previous two stages as input and performs an intra-procedural analysis. The only exception is WolfSSL, which has the most status codes. It needs considerably more time to check whether a presumed mapping is valid.

> Answer to RQ1: TRANSCODE is efficient in analyzing real-world programs. It can scale to projects with millions of lines of code and can finish the analysis in twenty-one minutes.

C. Results of Domain Inference

Ground Truth. The ground truth of all domains for five applications is manually collected. We first refer to official documents, if any, that describe the status codes' functionality [26, 27]. The rest of the unspecified status codes are classified manually according to whether they have related functionality in the source code.

Metric of Correctness. The correctness of domain inference is measured by how many domains match the ones in the ground truth. In the program source code, not all defined status codes are used and returned. Hence, the matching is defined by a subset of operations as follows:

$$\frac{\mathcal{D}_i \subseteq \mathcal{D}_j \qquad \mathcal{D}_i \in Domain, \mathcal{D}_j \in Ground\ Truth}{Match \leftarrow \mathcal{D}_i} \tag{4}$$

In the above formula, $Domains$ denotes the superset of all domains inferred by TRANSCODE, $Ground\ Truth$ are manually-collected domains of each program for five applications, and $Match$ denotes all inferred domains that match the ground truth.

Definition VI.1. *Match Rate* is defined as:

$$Match\ Rate = \frac{\#Match}{\#Domain} \tag{5}$$

The experimental results are listed in Table IV. In total, TRANSCODE has inferred 228 domains in five programs, and 218 of them match with the ground truth, achieving a match rate of 95.6%. The results also suggest inferred domains are capable as inputs to error detection of TRANSCODE.

The main source of mismatches is when a small number of functions do not follow the convention to produce status codes within the same domain. For example, Figure 6 shows a function that returns status codes from two domains from

TABLE IV
Results of Domain Inference

Project	Matches	Inferred	Match Rate
WolfSSL	20	24	0.833
Httpd	13	17	0.765
SQLite	76	78	0.974
GRPC	14	14	1.000
Asterisk	95	95	1.000
Total	218	228	0.956

```
1  static int authn_cache_post_config(...)
2  {
3      if (!configured)
4          return OK;
5      if (socache_provider == NULL)
6          return 500; // An HTTP status would be a misnomer!
7  }
```

Fig. 6. Example of outputting status codes from multiple domains

the ground truth. The status code `OK` returned at Line 4 refers to a group of status codes only internally used in Httpd, and status code `500` at Line 6 is an HTTP status code. As such, TRANSCODE allows both status codes to be grouped, i.e., these two codes, which in fact belong to two domains, are classified into one domain. Line 4 is an unfixed defect as the developer's comments indicate: *An HTTP status would be a misnomer*, meaning that HTTP status codes should never be returned here. Applications like SQLite and WolfSSL also have a few functions that return status codes of multiple domains, causing mismatches.

The results also imply that developers usually differentiate among status codes in a program when implementing different features. For example, Httpd outputs `HTTP_OK` to the clients when the HTTP request is valid, and signals `HTTP_BAD_REQUEST` when invalid. In either case, it never outputs status codes unrelated to HTTP operations, like `FTP_ERROR`, which cannot be interpreted and handled correctly by its users.

> Answer to RQ2: Overall, 95.6% of domains inferred by TRANSCODE match the ground truth.

D. Effectiveness of Mapping Error Detection

In this section, we evaluate the performance of TRANSCODE in detecting mapping errors on five real-world applications.

Overall Results. We apply TRANSCODE to 5 real-world applications: WolfSSL, Httpd, SQLite, Google RPC, and Asterisk. Table V lists numbers of mapping error reports. Regarding the number of bug reports, at least two authors participate in the bug confirmation[6]. In general, TRANSCODE reports 60 status code mapping errors in total, among which 59(=42+4+13) errors have been confirmed by ourselves, and we also find one report is a false positive. All confirmed reports have been submitted to developers via issue trackers or forums. The row *validated* denotes the numbers of confirmed bugs that have not been fixed. Thus far, 13 mapping errors have been fixed by developers or patches proposed by ourselves, and 4 reports are under review.

[6]Most authors agreed that all reports are true reports, so we do not adopt Kappa score to measure the quality.

TABLE V
Results of Mapping Error Detection

Projects	Reports	False	Confirmed		
			Validated	Reviewing	Fixed
WolfSSL	31	1	22	0	8
Httpd	7	0	5	0	2
SQLite	9	0	9	0	0
Asterisk	4	0	4	0	0
GRPC	9	0	2	4	3
Total	60	1	42	4	13

```
1   static int read_request_line(...){
2       ...
3       rv = ap_rgetline(...);
4       if (APR_STATUS_IS_ENOSPC(rv)) {
5           return HTTP_REQUEST_URI_TOO_LARGE;
6       }
7       else if (APR_STATUS_IS_TIMEUP(rv)) {
8           return HTTP_REQUEST_TIME_OUT;
9       }
10  +   else  if  (APR_STATUS_IS_BADARG(rv))
11  +       return HTTP_BAD_REQUEST;
12  +   }
13      else if (APR_STATUS_IS_EINVAL(rv)) {
14          return HTTP_BAD_REQUEST;
15      }
16      ...
17      return HTTP_OK;
18  }
```

Fig. 7. Improper mapping of a status code `APR_BADARG`. The green code is the developers' patch of this bug.

Confirmed Errors in Apache Httpd. TRANSCODE identified 7 mapping errors in Apache Httpd, where 2 of them have been fixed by the developers. Figure 7 shows an example of mapping errors[7][8]. The variable `rv` saves the status codes from callee function *ap_rgetline*. After that, it only maps three status codes `APR_ENOSPC`, `APR_TIMEUP` and `APR_EINVAL` and returns the corresponding HTTP status codes. TRANSCODE found another mapping also receiving status codes from *ap_rgetline()* but performing a mapping `APR_BADARG` $\mapsto$ `HTTP_BAD_REQUEST`, inconsistent to this example. Then, there exists an inconsistency of mapping the same code to two different ones. Apparently, `APR_BADARG` in the example is silently discarded and directly mapped to `HTTP_OK`, leaving the hidden errors buried. The developer also expressed the difficulty in determining which status code should be mapped for these status codes without strong prior knowledge.

Confirmed Errors in Google RPC. TRANSCODE identified 9 new bugs in Google RPC, where 3 have been fixed by its developers. In Figure 8, the developers forgot to map the status codes in variable `status` to a status code `GRPC_ERROR_NONE`

[7]https://bz.apache.org/bugzilla/show_bug.cgi?id=63669
[8]https://svn.apache.org/viewvc?view=rev&rev=1873394

```
1  static grpc_error add_socket_to_server(...) {
2      status = WSAIoctl(...);
3      if (status != 0) {
4  -       return NULL;
5  +       return GRPC_ERROR_NONE;
6      }
7      ...
8  }
```

Fig. 8. An incorrect status code mapping in Google RPC, where Lines 5-6 are the fix made by a developer.

rather than a NULL value. TRANSCODE identifies it because the same status codes are inconsistently mapped into `GRPC_ERROR_NONE` in another function *tcp_connect*. We reported two mappings to the developers, and they confirm that the second mapping is correct.

A False Positive in WolfSSL. Among all reported bugs in WolfSSL, one is regarded as a false positive, though an inconsistency of returned status codes is found by our tool[9]. The developer argued that the only functional status code is set to a field of a specific structure `ssl->err`, so there is no need to care about the returned status codes. While the report is marked as a false positive, it does not compromise our approach. One may still complement the semantics by considering the specific field as a destination in the mapping identification stage.

Reports Under Review. There are four reports of GRPC currently waiting for more information. To the best of our knowledge, the impact of mapping errors is not trivial to discover because it does not cause any visible effects such as crashes. However, to understand its real impact, expert-level domain knowledge is required to trigger the handling code of inconsistent mappings and monitor whether they lead to misbehaviors. Besides, even if these errors do not have immediate effects, they might mislead the future users of these functions and possibly leads to new bugs.

> Answer to RQ3: TRANSCODE reports 60 mapping errors among five real-world large-scale applications, of which 59 have been confirmed, and 13 have been fixed by developers.

E. Industry Experience

We have deployed TRANSCODE to large-scale microservice systems in WeChat, which process tens of millions of queries per day. TRANSCODE manages to find status code mapping errors[10]. These errors, nevertheless, are missed by the developers, regression testing, and industry-strength code analyzers in WeChat. Developers have fixed these errors to avoid the instability of microservice systems. We have received very positive feedback from the head of the development center for our high applicability in industrial settings.

[9]https://github.com/wolfSSL/wolfssl/issues/3213

[10]Due to a security contract, we are not permitted to disclose either details or the number of discovered bugs.

F. Threats to Validity

Threats to External Validity. The major threat to external validity is the representativity of evaluated subjects. The programs in the experiment might not be general enough to expose mapping behaviors. TRANSCODE's implementation may not suit some applications with other status code designs or merely using exceptions. However, we argue that these applications are representatives of different software eras in both C and C++. Besides, according to previous studies [28, 29], constraint solvers can also fail and report false results in some cases. We adopt the latest stable version of Z3 SMT Solver (v4.8.2) at the time we implement TRANSCODE to mitigate possible negative effects.

Threats to Internal Validity. There are mainly two factors that impact the practicality of mapping error detection.

The first threat is that the oracle for identifying mapping errors might not hold and be adequate at all places. More specifically, it is possible on some projects that one status code is considered valid to be mapped into two different ones under different conditions. However, the problem might bring a few false positives as we have not encountered that in our evaluated subjects. In addition, merely detecting via inconsistencies might mean some rare mapping errors would be missed if the only mapping is the bad mapping or mappings are all incorrect.

The second threat is the soundness of the status code propagation itself. We currently model all constant integers that can propagate to return statements as status codes. It is possible that status codes in some projects are wrapped in other types, like strings. To this end, one can easily extend our algorithm by considering type semantics.

VII. DISCUSSION

Identifying Status Code Mappings. Earlier propagation-based techniques [1, 2] detect missing-checks of status codes. Existing error-handling bug detectors [4, 5] merely concentrates on whether status codes are in the proper ranges. However, in fact, as pinpointed by [30], status codes are not general, and are divided into idiosyncratic sets of them. According to Gunawi et al. [1], one status code value may be mapped to another, typically from one domain to another. This is in order to exhibit its different functionality, in which case their solution cannot detect their errors of them. Numerous existing efforts mention these complex characteristics, yet none of them really make an effort to solve the problems among idiosyncratic sets of status codes. Thus, we present the first work to detect status code mapping errors in real-world systems.

Identifying Domains of Status Codes. Status codes are commonly divided into domains in real-world software, as observed in many software documents when working on different program parts [10, 27]. However, status codes are defined in various ways, making it hard to guess which domains they belong to. For many large-scale projects, status codes are mingled in different structures, involving pure magic integers,

enumerate types, and a series of macros, located in different files and classes. It is impossible to infer domain semantics from their definitions without a close look at their use patterns. For instance, most status codes of Httpd are defined in the same header `httpd.h`, but there exists no regular indicator for which status codes represent the related functionality. Instead of extracting domains from definitions, we propose to infer them directly from uses.

Confirming Mapping Error Reports. The long reviewing period of submitted reports might lead to an underestimation of the impact of mapping errors. Some issues have not received responses from developers after several months. One of the reasons might be the complexity of reports for mapping errors. For each error report, since mapping errors are composed of at least two inconsistent mappings, the developer needs to manually track down the propagation of more than two status codes across several files. Nevertheless, to our knowledge, we explained in detail how these errors occur when reporting them. Therefore, a possible future direction is to automatically generate an explanation of reports that help developers to understand a mapping error.

Extensions to Other Error-handling Mechanisms. Our status code mapping error detection establish mainly on languages that mainly adopts status codes as their error-handling mechanisms. There also exists others, e.g., *exceptions* in Java and Python. In particular, a similar term "rethrow" of exceptions to status code mappings, denoting throw another exception during the handling of an exception, is studied in previous work [31]. A similar static analysis can be applied to check whether two rethrown exceptions are valid under the same context. One may implement a type inference and data-flow analysis to capture the exception flows [32] and apply our domain-based method to detect mapping errors.

VIII. Related Work

Error Propagation Analysis. Existing methods track the error codes to detect error propagation bugs. Gunawi et al. [1] track POSIX status codes over call graphs to detect the propagation bugs. Rubio-González et al. [2] use an inter-procedural data-flow analysis to analyze error codes of file systems. Weiss et al. [30] scale this method to large systems by storing the error propagation paths in a database. There are also statistical methods [33–35] infer the probabilities of error propagation.

All these methods only analyze the propagation of predefined error codes, while TRANSCODE analyzes the propagation of all status codes, a more general and broader problem.

Mining Error-handling Specifications. There are many approaches to inferring error-handling specifications. Weimer and Necula [36] mine the code for temporal specifications and detect error-handling bugs by verifying typestate properties. Acharya and Xie [37] automatically mine API error-check and cleanup specifications from the source code. A later work, FUNC2VEC [38] improves the quality of API specifications by mining static program traces generated from random walks of the inter-procedural control flow graph of the program. Eberhardt et al. [39] leverage unsupervised learning to learn API aliasing specifications, which could be further used to detect error-handling errors. Provided with domain knowledge, DeFreez et al. [6] and Zhou et al. [40] can generate error specifications via error path inference. Machine learning techniques [7] and probabilistic methods [41] are also used to predict error paths and mine error specifications. Type inference methods [42, 43] infer type comparable variables, and thus can be used to detect inconsistencies of program operations.

Our approach uses an inter-procedural value-flow analysis for inferring status code specifications, which lead to more fined-grained, and more precise error specifications being discovered.

Detecting Error-handling Bugs. Rubio-González and Liblit [44] claim the bugs if they find inconsistent handling to the same error codes between documentation and the source code. Weimer and Necula [3] detect error-handling bugs by tracing the propagation of unchecked exceptions in Java applications. EPEx [4] leverages the error specifications from APEx [41] to detect error-handling bugs via symbolic execution. ErrDoc [5] improves EPEx to detect wider categories of error-handling bugs and generate bug-fixes automatically.

Although previous techniques might detect few bugs overlapped with those detected by TRANSCODE, in general, status code mapping errors, consisting of more complex semantic information, are much harder to detect. TRANSCODE can automatically detect them without relying on extra knowledge, for example, what are the errors and non-errors. Besides, TRANSCODE employs static program analysis instead of symbolic execution. Thus, it scales to large-scale software systems.

IX. Conclusion

Mapping status codes is a common programming pattern in software development. To the best of our knowledge, we are the first to focus on detecting errors in status code mappings. In this paper, we propose TRANSCODE to automatically detect such errors in large software projects. It is evaluated on five real-world software projects. The results show TRANSCODE is promising in detecting status code mapping errors: it has discovered *59* mapping errors in five projects, *13* of which have been fixed by the developers.

Acknowledgement

We thank the anonymous reviewers for their insightful comments. We also appreciate engineers from Tencent in assisting system integrations. Rongxin Wu is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001) and NSFC 61902329. Other institutional authors are supported by the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, and the donations from Microsoft and Huawei.

References

[1] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. Eio: Error handling is

occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST 2008, pages 14:1–14:16, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1364813.1364827.

[2] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2009, pages 270–280, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542506. URL http://doi.acm.org/10.1145/1542476.1542506.

[3] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 419–431, 2004. doi: 10.1145/1028976.1029011. URL https://doi.org/10.1145/1028976.1029011.

[4] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 345–362, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana.

[5] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 752–762, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106300. URL http://doi.acm.org/10.1145/3106237.3106300.

[6] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 466–476, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338960. URL http://doi.acm.org/10.1145/3338906.3338960.

[7] Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. Generating precise error specifications for c: A zero shot learning approach. *Proc. ACM Program. Lang.*, 3(OOPSLA):160:1–160:30, October 2019. ISSN 2475-1421. doi: 10.1145/3360586. URL http://doi.acm.org/10.1145/3360586.

[8] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. Ufo: Verification with interpolants and abstract interpretation. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, page 637–640, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642367410. doi: 10.1007/978-3-642-36742-7_52. URL https://doi.org/10.1007/978-3-642-36742-7_52.

[9] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, page 123–136, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 354037406X. doi: 10.1007/11817963_14. URL https://doi.org/10.1007/11817963_14.

[10] Apple Developer. Error domain, 2021. URL https://developer.apple.com/documentation/foundation/nserror.

[11] Paul R. Halmos. *Naive Set Theory*. Springer New York, New York, NY, 1974. ISBN 978-1-4757-1645-0. doi: 10.1007/978-1-4757-1645-0_8. URL https://doi.org/10.1007/978-1-4757-1645-0_8.

[12] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 693–706, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192418. URL http://doi.acm.org/10.1145/3192366.3192418.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 270–280, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375615. URL https://doi.org/10.1145/1375581.1375615.

[14] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22 (3):269–285, 1957. ISSN 00224812. URL http://www.jstor.org/stable/2963594.

[15] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 103–114, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136625. doi: 10.1145/781131.781144. URL https://doi.org/10.1145/781131.781144.

[16] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 145–157, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138075. doi: 10.1145/996841.996860. URL https://doi.org/10.1145/996841.996860.

[17] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 131–144, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138075. doi: 10.1145/996841.996859. URL https://doi.org/10.1145/996841.996859.

[18] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. ISBN 0769521029. doi: 10.5555/977395.977673. URL http://doi.acm.org/10.5555/977395.977673.

[19] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[20] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 480–491, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250789. URL http://doi.acm.org/10.1145/1250734.1250789.

[21] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4241-4. doi: 10.1145/2892208.2892235. URL http://doi.acm.org/10.1145/2892208.2892235.

[22] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. 48(6): 435–446, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462159. URL https://doi.org/10.1145/2499370.2462159.

[23] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 77–101, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540601600. doi: 10.5555/646153.679523. URL http://doi.acm.org/10.5555/646153.679523.

[24] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE 2011, pages 343–353, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025160. URL http://doi.acm.org/10.1145/2025113.2025160.

[25] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[26] Google. Status codes and their use in grpc, 2021. URL https://github.com/grpc/grpc/blob/master/doc/statuscodes.md.

[27] Apache. Apr error space, 2021. URL https://apr.apache.org/docs/apr/1.6/group___a_p_r___e_r_r_o_r__map.html.

[28] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Skeletal approximation enumeration for smt solver testing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021. Association for Computing Machinery, 2021. doi: 10.1145/3468264.3468540.

[29] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing smt solvers via two-dimensional input space exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 322–335, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384599. doi: 10.1145/3460319.3464803. URL https://doi.org/10.1145/3460319.3464803.

[30] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis for scalable error propagation. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 586–597, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818827.

[31] Chen Fu and Barbara G Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proceedings - International Conference on Software Engineering*, pages 230–239, 2007. ISBN 0769528287. doi: 10.1109/ICSE.2007.35.

[32] Martin P. Robillard and Gail C Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, 2003. ISSN 1049331X. doi: 10.1145/941566.941569.

[33] Xiaocheng Ge, Richard Paige, and John McDermid. Probabilistic failure propagation and transformation analysis. pages 215–228, 09 2009. ISBN 978-3-642-04467-0. doi: 10.1007/978-3-642-04468-7_18.

[34] Petar Popic, D. Desovski, Walid Abdelmoez, and Bojan Cukic. Error propagation in the reliability analysis of component based systems. volume 2005, pages 10 pp.–, 12 2005. ISBN 0-7695-2482-6. doi: 10.1109/ISSRE.2005.18.

[35] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *2001 International Conference on Dependable Systems and Networks*, pages 161–170, July 2001. doi: 10.1109/DSN.2001.941402.

[36] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31980-1. doi: 10.1007/978-3-540-31980-1_30.

[37] Mithun Acharya and Tao Xie. Mining api error-handling specifications from source code. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, pages 370–384, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00593-0. doi: 10.1007/978-3-642-00593-0_25.

[38] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 423–433, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236059. URL https://doi.org/10.1145/3236024.3236059.

[39] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. Unsupervised learning of api aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 745–759, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314640. URL https://doi.org/10.1145/3314221.3314640.

[40] Chunjie Zhou, Xiongfeng Huang, Xiong Naixue, Yuanqing Qin, and Shuang Huang. A class of general transient faults propagation analysis for networked control systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(4):647–661, April 2015. ISSN 2168-2232. doi: 10.1109/TSMC.2014.2384480.

[41] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 472–482, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970354. URL http://doi.acm.org/10.1145/2970276.2970354.

[42] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. doi: 10.1016/j.scico.2007.01.015. URL https://doi.org/10.1016/j.scico.2007.01.015.

[43] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, page 338–348, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919149. doi: 10.1145/253228.253351. URL https://doi.org/10.1145/253228.253351.

[44] Cindy Rubio-González and Ben Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, page 73–80, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300827. doi: 10.1145/1806672.1806687. URL https://doi.org/10.1145/1806672.1806687.