Client-Specific Equivalence Checking

Yi Li[1], Federico Mora[1], Julia Rubin[2], and Marsha Chechik[1]

[1] University of Toronto, {liyi,fmora,chechik}@cs.toronto.edu
[2] University of British Columbia, mjulia@ece.ubc.ca

Abstract. Software is often built by integrating components created by different teams or even different organizations. Changes in a library component may trigger a sequence of updates for downstream clients. As such, developers often delay upgrading libraries, negatively affecting correctness and robustness of their systems. In this paper, we investigate the effect of component changes on the behavior of the component's downstream clients. We observe that changes in a library are often irrelevant to a particular client and thus can be adopted without any delays or negative effects. Following this observation, we formulate the notion of *client-specific equivalence checking*, develop an automated technique for checking such equivalence, evaluate the technique on a set of benchmarks, and show its applicability and effectiveness.

1 Introduction

Large software systems are often composed of multiple related but independently-developed software components. Specifications of such components are usually limited to the description of APIs and do not include implicit internal assumptions. Moreover, specifications regularly become obsolete. With little or no information about the internals of (library) components, it becomes challenging to understand the semantic implications of their evolution on downstream client components [20]. An empirical study of library updates [15] shows that 81.5% of the studied systems keep outdated library dependencies, primarily due to the significant effort required to migrate to new versions.

There are several existing techniques for validating behavioral equivalence between two versions of a program or for identifying the precise set of changes between them, e.g., *differential symbolic execution* [18] and *differential assertion checking* [16]. Yet, these equivalence checking techniques are limited in the sense that they do not consider the *usage pattern* of a particular library component within its client. We postulate that the equivalence checking problem becomes both more relevant and more tractable when the usage pattern is considered. In particular, reasoning about a library update w.r.t. its impact on the client allows us to make more informed decisions on whether a library migration is safe from the view of a particular client. Also, by considering library usage patterns in specific clients, we only need to analyze library components within specific contexts which may significantly reduce the complexity of equivalence checking, enabling us to produce a highly efficient and accurate analysis.

Our preliminary analysis of 60 real-life clients of four large open-source libraries showed that, in 68% of the cases, the behavior of the client component

```
1  double mpf_get_d_2exp (
2      signed long int *expptr,
3      mpf_srcptr src) {
4    mp_size_t size, abs_size;
5    mp_srcptr ptr;
6    int cnt;
7  +  double d;
8
9    size = SIZ(src);
10   if (UNLIKELY (size == 0)) {
11     *expptr = 0;
12     return 0.0;
13   }
14
15   ptr = PTR(src);
16   abs_size = ABS(size);
17   count_leading_zeros(cnt, ptr[abs_size
     ↪  - 1]);
18   cnt -= GMP_NAIL_BITS;
19
20   *expptr = EXP(src) * GMP_NUMB_BITS -
     ↪  cnt;
21
22 -  return  mpn_get_d(ptr, abs_size, 0,
23 -    -(abs_size * GMP_NUMB_BITS - cnt));
24 +  d = mpn_get_d(ptr, abs_size, 0,
25 +    -(abs_size * GMP_NUMB_BITS - cnt));
26 +  return size >= 0 ? d : -d;
27  }
```

(a) Simplified patch *#17323* from the GMP library.

```
1 double F_mpz_poly_eval_horner_d_2exp(
2   long * exp, F_mpz_poly_t poly, double
    ↪  val) {
3   ...
4   res = mpf_get_d_2exp(exp, output);
5   // work around bug in earlier versions
    ↪  of GMP/MPIR
6   if ((mpf_sgn(output) < 0) && (res >=
    ↪  0.0))
7     res = -res;
8   ...
9 }
```

(b) Client function *F_mpz_poly_eval_horner_d_2exp* from the FLINT library [9].

```
1 REAL log_real(REAL x) {
2   double d;
3   double ln_app;
4   signed long int exp;
5
6   d = mpf_get_d_2exp(&exp,
    ↪  x.get_mpf_t());
7   ln_app = (double) exp *log(2.0) +
    ↪  log(d);
8   return ln_app;
9 }
```

(c) Client function *log_real* from the MPACK library [17].

Fig. 1. An update and two sample clients of the *mpf_get_d_2exp* library function.

is completely unaffected by changes in its libraries (see Appendix A for more details). Fig. 1a shows one such example: a simplified library patch extracted from change histories of the GNU Multiple Precision Arithmetic Library (GMP) [10]. The function *mpf_get_d_2exp* in the source file *mpf/get_d_2exp.c* converts a GMP floating-point number (input parameter *src*) to a double (return value *d*), with an exponent returned separately (output parameter *expptr*). The input number *src* always has the same sign as the variable *size* (updated on line 9). Before the changes are introduced, the function always returns a positive number (lines 22-23), whereas after the changes, the return value *d* has the same sign as the input (lines 24-26). The documentation[3] is ambiguous about the sign of the return value and this causes confusion[4] among developers who use this library.

Fig. 1b shows a client that uses the *mpf_get_d_2exp* function from Fig. 1a. The function *F_mpz_poly_eval_horner_d_2exp* reverses the sign of the returned value only when it should indeed be reversed (lines 6-7). Therefore, the patch in lines 22-26 of Fig. 1a does not affect the behavior of this client. In contrast, the client in Fig. 1c is affected as it does not check the sign of the returned value but rather passes it directly to *log(double)* whose behavior is undefined on negative

[3] See `https://gmplib.org/manual/Converting-Floats.html`.

[4] See `https://gmplib.org/list-archives/gmp-bugs/2017-March/004112.html` for a developer conversation on this topic.

```
1 int client1(int x) {
2   if (x > 10)
3     return x;
4   else
5     return lib1(x);
6 }
```

(a) *client1*.

```
1 int lib1_old(int x) {
2   return x;
3 }
```

(b) *lib1_old*.

```
1 int lib1_new(int x) {
2   if (x > 10)
3     return 9;
4   else
5     return x;
6 }
```

(c) *lib1_new*.

```
1 int client2(int x) {
2   if (x > lib2(x))
3     return x;
4   else
5     return lib2(x);
6 }
```

(d) *client2*.

```
1 int lib2_old(int x) {
2   return x - 1;
3 }
```

(e) *lib2_old*.

```
1 int lib2_new(int x) {
2   return x;
3 }
```

(f) *lib2_new*.

Fig. 2. Client and library pairs illustrating two types of client-specific equivalence.

inputs (line 7). Thus, the behavior of this client, in situations when the input parameter x is negative, differs before and after the patch to *msf_get_d_2exp*.

In this paper, we aim to provide a method for differentiating between these two cases. We formalize the notion of *Client-Specific Equivalence* (CSE): equivalence of library component versions w.r.t. a particular client, and provide implementation for efficiently establishing CSE. More specifically, this paper contributes: (1) empirical evidence for the prevalence of library changes that do not affect individual clients and, consecutively, for the applicability of our approach in practice; (2) a generic framework for checking CSE parameterized by different notions of *equivalence criteria* defined over client behaviors; (3) an *incremental lazy exploration* technique that improves the effectiveness of the generic framework; (4) concrete implementation of both client-specific equivalence checking techniques, called Freud and Freud+; (5) an empirical evaluation of the effectiveness of Freud and Freud+.

The rest of the paper is structured as follows. Sec. 2 presents the overview of our approach on simple examples. Sec. 3 fixes the notation and provides the necessary background. We formally define our client-specific equivalence checking framework in Sec. 4. In Sec. 5, we describe the implementations of our approach and evaluate their effectiveness. Sec. 6 discusses related work. We conclude in Sec. 7 with a summary and possible future directions.

2 Examples and Overview

In this section, we give an overview of our approach for determining equivalences of libraries with respect to a particular client on two simple examples. Fig. 2 shows two client-library pairs, namely, *client1* which depends on *lib1* (Figs. 2a-c), and *client2* which depends on *lib2* (Figs. 2d-f). Each of the libraries have two versions, "*old*" and "*new*", which are equivalent in the eyes of the client. For simplicity, we assume that all variables take arbitrary input values from the unbounded integer domain, so there are no overflows or underflows.

Example 1. Fig. 2a shows the source code of a client program; Figs. 2b and 2c show the two versions of a library on which this client could depend. The change introduced in the new version is an if-statement which splits the single program path of the old version into two: when the input value is greater than 10 and when it is not. In this example, the behavior of *lib1_old* and *lib1_new* is different for any input $x > 10$: the old library will return x and the new library will always return a constant 9. Yet, the two library versions are equivalent in the eyes of *client1* because the library is never called with $x > 10$ (lines 2-4 in Fig. 2a). As such, the change in *lib1_new* is never exercised. In fact, the two library versions are *conditionally equivalent* [13] under the condition $x \leq 10$ and both return x for any given input x. Since *lib1* is only called by the client under such a condition (line 5 in Fig. 2a), we say that that library versions are *client-specific equivalent* w.r.t. *client_1*.

Example 2. Fig. 2d shows another client, *client2*. Figs. 2e and 2f show two versions of the library it calls. The only change to the old library version is the replacement of "$x - 1$" by "x". Although the two library versions are obviously not equivalent, the difference does not lead to different client behavior: *lib2_old* always returns a value which is smaller than the input x, leading the *client2* into the if-branch (lines 2-3 in Fig. 2d). As the result, the input value x is returned by the client. The new version of the library, *lib2_new*, returns the input itself, leading the execution of *client2* into the else-branch (lines 4-5 in Fig. 2d); yet, input value x is returned by the client in this case as well.

Unlike Example 1, here the change in the library does get exercised by the client. The two library versions are still client-specific equivalent though: the change on the library is "digested" by the client so that the final output is unaffected. The example shown earlier in Fig. 1b falls into the same category.

Equivalence Criteria. While in both examples the changes in the library programs do not affect the final return values of the client, the intermediate execution of the client might be altered. Based on this, we characterize two kinds of *equivalence criteria* defined in terms of the client's behaviors:

Functional equivalence: if we ignore the internal execution steps and only observe the input and output values, the clients in both examples are functionally equivalent, since they all return the same values given same inputs.

Path equivalence: if we consider the sequence of commands executed and the program locations visited during each run, only the client in Example 1 is path-equivalent as all intermediate execution steps get preserved. Example 2 does not satisfy path equivalence criteria as *client2* has different behaviors before and after the library change: visiting $4 \rightarrow 5$ rather than $2 \rightarrow 3$ in Fig. 2d.

These equivalence criteria are meaningful in practice. Functional equivalence focuses only on the final outputs of the client and thus can be used to gauge the effects of library changes on external users of the client. Path equivalence requires that all the intermediate computation steps are preserved, and thus it also preserves side-effects.

Checking CSE. Now, we describe our general framework for determining client-specific equivalence. Fig. 3 overviews the architecture of the framework. The framework accepts as input a client and two versions of a library. We assume no change on library interfaces, since they can often be easily caught by compilers. The framework's main components are a *behavior explorer* and an *equivalence verifier*. The behavior explorer is parameterized by equivalence criteria – either functional or path equivalence. With a given equivalence criterion, we then generate *equivalence assertions* that must be satisfied for the libraries to be CSE. Finally the equivalence verifier checks the libraries against the equivalence assertions and either declares them to be equivalent or provides a counter-example which demonstrates the behavioral differences observed in the client.

Incremental Lazy Exploration. The approach outlined above is not efficient because it fully explores both the client and library programs before generating equivalence assertions based on the explored behaviors. Furthermore, it is often not feasible to exhaustively explore all program behaviors and thus arrive at a definitive conclusion.

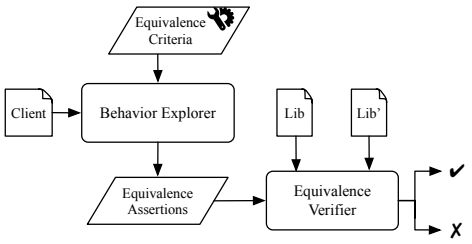

Fig. 3. Architecture of the client-specific equivalence checking framework.

Comparing Examples 1 and 2, we identify two types of CSE: (1) *inactive CSE* – when changes are not exercised by the client and (2) *active CSE* – when changes are anticipated, exercised, and specially handled by the client. The biggest distinction between the two is that to show inactive CSE, one does not need to reason about the semantics of the libraries: a purely syntactic check on all feasible library paths suffices to confirm that no change gets exercised. On the other hand, to claim active CSE, one needs to analyze the input-output relations of the libraries and their interplay with the client.

With this insight, we propose a CSE checking approach called *incremental lazy exploration* which prioritizes establishing proof arguments for inactive CSE while keeping the number of paths explored to a minimum. Every time when a path containing a call to the library is explored, we effectively obtain a *client context* for the library call. With this context we perform a *parallel exploration* of both library versions under the given context by shadowing [4] one with the other. This step produces a set of library path pairs, and for each pair, two checks are performed: (1) is the path exercising changed (added and removed) code? (2) does a concrete simulation on the path pair reveal a counterexample? Both of these checks are relatively cheap; check (2) can immediately reveal a counterexample for CSE; and if check (1) fails for all client contexts, then we can also conclude CSE without reasoning about the actual semantics of the paths explored.

	Partition	Effect	Explored	Active
1	$x > 10$	$RET = x$	Yes	No
2	$\neg(x > 10)$	$RET = lib1(x)$	Yes	Yes

(a) *client1.*

	Partition	Effect	Explored	Active
1	True	$RET = x$	Yes	No

(b) *lib1_old.*

	Partition	Effect	Explored	Active
1	$\neg(x > 10)$	$RET = x$	Yes	No
2	$x > 10$	$RET = 9$	No	No

(c) *lib1_new.*

Fig. 4. Paths explored for Example 1.

In Example 1, we begin by exploring the client and skip the first path which does not involve library calls (Row 1 in Fig. 4a). Then we explore the second path, shown in Row 2 of Fig. 4a, and perform a parallel exploration on both libraries with the client context $\neg(x > 10)$. The first rows of Figs. 4b and 4c show the only pair of library paths explored. We then perform the first check and realize that the change is not active on this path; therefore, upon finishing the library exploration, we can conclude that the libraries are equivalent under the current client context. Since no other client paths remain, we identify the case of inactive CSE. We evaluate the relative efficiency of incremental lazy exploration in Sec. 5.

3 Background

In this section, we provide the necessary background on programs and program analysis that will be used in the remainder of the paper.

Programs. We restrict the presentation to a simple imperative programming language where all operations are either assignments, assumptions or procedure calls, and all variables range over integers. Without loss of generality, we assume that the type and number of input and output parameters are statically known for each procedure.

A program $P = (F_c, \{F_l\}_i)$ consists of a client and a set of library procedures, such that the client calls the libraries. Each of the procedures can be represented as a control flow graph $(\mathcal{L}, l_0, l_f, E, \mathcal{V})$, where $\mathcal{L}$ is a finite set of program locations, with an initial location $l_0 \in \mathcal{L}$ and a final location $l_f \in \mathcal{L}$. The set $\mathcal{V}$ denotes a finite set of variables, and $E \subseteq \mathcal{L} \times \Sigma \times \mathcal{L}$ is the set of control-flow edges, where Σ is the set of operations instantiated by one of the following constructs: (1) an assignment $v \leftarrow exp$, where $v \in \mathcal{V}$; (2) an assumption of the form *assume(b)*, where b is a Boolean expression over program variables $\mathcal{V}$; (3) client is allowed to make procedure calls to library procedures, e.g., $\boldsymbol{x} \leftarrow F(\boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of variables in $\mathcal{V}$ and F is a procedure in $\{F_l\}_i$.

We write $l \xrightarrow{\sigma} l'$ instead of $(l, \sigma, l') \in E$ to denote an edge from l to l' introduced by an operation $\sigma \in \Sigma$. We assume that all executions of P terminate, but this assumption does not prevent P from possibly having infinite number of paths, such as in the case where there is a loop whose number of iterations depends on an unbounded variable.

$$[\![\cdot]\!] : (\mathcal{V} \mapsto \mathbb{Z}) \mapsto (\mathcal{V} \mapsto \mathbb{Z})$$

$$[\![x \leftarrow exp]\!](v) = v[x \mapsto [\![exp]\!]_v] \qquad \text{assignment}$$

$$[\![assume(b)]\!](v) = \begin{cases} v & \text{if } v \models b \\ \bot & \text{otherwise} \end{cases} \qquad \text{assume}$$

$$[\![\boldsymbol{x} \leftarrow F(\boldsymbol{y})]\!](v) = [\![assume(\varphi_F[\boldsymbol{x}/\boldsymbol{\alpha}, \boldsymbol{y}/\boldsymbol{\beta}])]\!]_v \qquad \text{procedure call}$$

Fig. 5. Semantics of operations.

Procedure Summaries. Given a program P and a procedure F_i, the *procedure summary* of F_i is a first-order formula φ_i over vectors of variables $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ such that $\boldsymbol{\alpha}$ denotes F_i's input parameters and $\boldsymbol{\beta}$ denotes its outputs. A procedure summary is *complete* if it is defined for all possible inputs. We write $P[\varphi_i]$ to denote the program with every procedure call $\boldsymbol{x} \leftarrow F_i(\boldsymbol{y})$ replaced by $assume(\varphi_i[\boldsymbol{x}/\boldsymbol{\alpha}, \boldsymbol{y}/\boldsymbol{\beta}])$. In other words, $P[\varphi_i]$ is like P but with every procedure call F_i replaced by its function summary.

Concrete, Abstract, and Observable Runs. We interpret program executions using a Labelled Transition System (LTS), $(\mathcal{S}, s_0, s_f, \Sigma, \rightarrow)$ for each control-flow graph, where $\mathcal{S} = \mathcal{L} \times (\mathcal{V} \mapsto \mathbb{Z})$ is a set of program states, and $s_0 \in \mathcal{S}$ and $s_f \in \mathcal{S}$ are the initial and final states, respectively. Let $v : \mathcal{V} \mapsto \mathbb{Z}$ be a valuation of the variables at state s. We write $(l, v) \xrightarrow{\sigma} (l', v')$ to represent the transition from state s to s' if $l \xrightarrow{\sigma} l'$ and $(v, v') \in [\![\sigma]\!]$ (defined in Fig. 5). A *concrete run* $\pi = s_0 \xrightarrow{\sigma_0} \cdots \xrightarrow{\sigma_k} s_k$ of an LTS is an execution path that starts with an initial state. The set of all concrete runs is written as Π. An *abstract run* is a set of concrete runs $\hat{\pi} = \{\pi_i\}$. Let $\omega : \Pi \mapsto \Pi$ be an *observation function* which maps a concrete run π to its observable part $\omega(\pi)$.

Recall *client1* shown in Fig. 2a. With the values of x and RET written compactly as a tuple, $(2, (X, \top)) \rightarrow (3, (X, X))$ is an abstract run which subsumes all concrete runs going through the if-branch of *client1*, where $\top$ denotes an uninitialized value.

Symbolic Execution. *Symbolic execution* [14] uses *symbolic values* as input, instead of the actual data, and represents the values of program variables as symbolic expressions. As a result, output values computed by the program are also represented as symbolic expressions. The *state* of a symbolically-executed program includes the symbolic values of program variables, a path condition and a program location. The *path condition* is a boolean formula over the symbolic inputs corresponding to the accumulated constraints along the path which are satisfiable if and only if the associated path is feasible. A symbolic path corresponds to an abstract run of the program, which can be instantiated to a concrete run by computing a satisfying model of the path condition.

4 Our Approach

In this section, we first formalize our client-specific equivalence checking framework and then report on a specific instantiation which leverages common patterns observed in inactive CSE cases to speed up the checking process.

4.1 The CSE Checking Problem

Here we define a generic client-specific equivalence checking framework which can be parameterized by different types of client equivalence criteria.

Problem Definition. Let F_c be a client procedure, and let F_l and F_l' be two versions of a library procedure such that they share the same signature and can be called interchangeably from F_c. Let ω be an *observation function* which maps a concrete run to an *observable run*. Let $Behav(F)$ denote the set of all (concrete) runs of a procedure F.

Definition 1 (Client-Specific Equivalence). *We say F_l and F_l' are* client-specific equivalent *w.r.t. F_c and ω, denoted by $F_l \equiv_{(F_c,\omega)} F_l'$, if and only if there is a one-to-one mapping between $\{\omega(\pi)|\pi \in Behav(F_c[F_l])\}$ and $\{\omega(\pi')|\pi' \in Behav(F_c[F_l'])\}$.*

Client-specific equivalence of two library versions is defined in terms of the observable behaviors of the client. Two concrete paths π and π' are equivalent if they follow the exact same sequence of state transitions. Two procedures F and F' are considered equivalent when there is a one-to-one mapping between their sets of concrete runs. We say that the observable behaviors of F and F' are equivalent if they both have the same set of observable runs defined by ω. Finally, two libraries F_l and F_l' are client-specific equivalent when the observable behaviors of the composed programs, $F_c[F_l]$ and $F_c[F_l']$, are equivalent.

The most restrictive definition of CSE requires that each concrete run of the client is not altered by the library changes. This is effectively the path equivalence demonstrated in Example 1. On the other hand, functional equivalence illustrated in Example 2 can be defined by introducing an observation function which only considers the input and output values of the client and ignores all the intermediate transitions. A wide range of other equivalence criteria can be defined but in the rest of this paper, we focus on *functional CSE* where client behaviors are required to be functionally equivalent.

4.2 Checking Functional CSE with Symbolic Execution

In most cases, there is an infinite number of concrete runs and thus checking client-specific equivalence by enumerating all runs of a client (composed with both libraries) and explicitly comparing them against each other is often infeasible. We show that for certain families of equivalence criteria, it is possible to reduce the CSE checking problem to the validity of first-order formulas.

We now describe our algorithm Freud for checking functional CSE using symbolic execution. The inputs to the algorithm are a client, F_c, and two related libraries sharing the same interface, F_l and F_l'.

Behavior Exploration. First, Freud explores the behaviors of the client through symbolic execution without considering the bodies of the libraries. It does so via standard path exploration while replacing each library call with an uninterpreted function placeholder. Focusing only on the client program reduces the number of paths and allows for modular checking of libraries. The abstract runs returned from exploring the procedure can be represented symbolically as a

set of *partition-effect pairs* [18], namely, (pc_i, ob_i), where pc_i represents a *path constraint* and ob_i stands for an observable *effect constraint* for a particular path $\hat{\pi}_i$. The path constraint is a conjunction of relational expressions defined over constants and input variables. The effect constraint is a conjunction of expressions which equate a special return variable "RET" to expressions over constants and input variables. Representing effects as symbolic expressions over input variables allows us to reason about multiple concrete runs together.

The set of abstract runs returned from exploring a procedure is its *summary*. Summaries can be *incomplete* due to the limitations in symbolic execution. For example, with the presence of unbounded loops, it is only possible to get paths of limited length.

Equivalence Assertion Generation. Let $Behav(F_c[F_l])$ and $Behav(F_c[F_l'])$, be the summaries produced in the previous step. FREUD takes these and generates an *equivalence assertion* ϕ – a first-order formula with uninterpreted functions as placeholders for the libraries. The formula ϕ serves as a *mutual specification* [16] for the two libraries – the observable behaviors of the client are equivalent if and only if the library bodies respect the equivalence assertion. Following the notion of *logical method summary* [11], the observable behaviors of the client in this case can be encoded as a disjunction over all symbolic paths, and the resulting equivalence assertion is $\phi := \big(\vee_{\hat{\pi}_i \in Behav(F_c[F_l])} (pc_i \wedge ob_i) \iff \vee_{\hat{\pi}_i' \in Behav(F_c[F_l'])} (pc_i' \wedge ob_i')\big)$, where pc_i and ob_i are path constraint and effect constraint of $\hat{\pi}_i$, respectively.

Equivalence Verification. Finally, ϕ is verified against the library implementations F_l and F_l'. The verification task can be delegated to a theorem prover based on the procedure summaries of the libraries. We first generate procedure summaries for both library versions. The summaries for F_l and F_l' are then used in place of the uninterpreted function placeholders in the equivalence assertion ϕ. Finally, we check the validity of the equivalence assertion composed with library summaries. If a violation is found, we report a counterexample; otherwise, if the generated summaries are complete, we proved functional CSE.

Proposition 1. *Given complete summaries of F_c, F_l and F_l', the equivalence assertion is valid if and only if F_l and F_l' are client-specific equivalent w.r.t. F_c.*

Proof Sketch. Since all summaries are complete, any concrete run of the client is a model of the path constraints. From Def. 1, F_l and F_l' are client-specific equivalent since a one-to-one mapping between concrete runs is established by the logical equivalence between composed summaries.

4.3 Improving Freud with Incremental Lazy Exploration

We now present FREUD+, an optimized implementation of FREUD that uses *incremental lazy exploration* – a behavior exploration strategy which prioritizes establishing proof arguments for inactive CSE – see Sec. 2. A pseudo-code implementation of FREUD+ is given in Fig. 6 and the workflow for this technique in Fig. 7. We walk through the code and offer the workflow as a visual aid.

The inputs to the algorithm, as before, are a client, F_c, and two related libraries sharing the same interface, F_l and F_l'. This time, however, we are only

Require: F_c calls F_l and F_l' interchangebly
Ensure: If $F_l \equiv_{F_c} F_l'$ then returns true, else returns false

```
1: procedure FREUD+(F_c, F_l, F_l')
2:     A ← Initial                          ▷ Initialize active summaries of F_c, F_l, F_l'
3:     while p ∈ EXPLORE(F_c[f_l(x)]) do
4:         Behav_p(F_l) ← EXPLORE(F_l(x) | x ⊨ p)
5:         Behav_p(F_l') ← EXPLORE(F_l'(x) | x ⊨ p)          ▷ Summaries mod p
6:         if p uses change then
7:             if concrete value for p is a counterexample then return false
8:             A.UPDATE(p, Behav_p(F_l), Behav_p(F_l'))
9:         end if
10:    end while
11:    if EMPTY(A) then return true
12:    φ ← ASSERTGEN(A)
13:    if VERIFY(φ, F_l, F_l') then return true
14:    else return false
15: end procedure
```

Fig. 6. Algorithm for checking functional CSE based on symbolic execution and lazy path exploration.

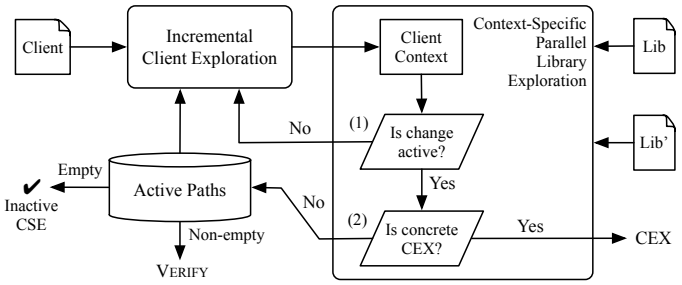

Fig. 7. FREUD+ work flow with incremental lazy exploration.

interested in exploring and reasoning about active paths. We call a summary consisting of only active paths an *active summary*.

The while loop at line 3 drives the lazy exploration: its body is executed until the client is fully explored. Each iteration takes a partition-effect pair, p, from the client's summary and processes the libraries modulo p. This results in $Behav_p(F_l)$ and $Behav_p(F_l')$ respectively. If these summaries exercise the difference in the library versions, checked at line 6, then FREUD+ does two things. First it finds a concrete value that satisfies p and checks if it is a counterexample. If it is a counterexample FREUD+ reports that F_l and F_l' are not client-specific equivalent. Second, FREUD+ updates the active summaries. If no active paths have been found after the client has been fully explored FREUD+ reports equivalence. Otherwise, FREUD+ will use the active summaries to generate an assertion, at line 12, and then check it, at line 13.

Proposition 2. *Let* complete active summary *mean the maximal active subset of a complete summary. Given complete active summaries of F_c, F_l and F_l', the equivalence assertion is valid if and only if F_l and F_l' are client-specific equivalent w.r.t. F_c.*

	Partition	Effect	Explored	Active
1	$x > lib2(x)$	$RET = x$	Yes	Yes
2	$\neg(x > lib2(x))$	$RET = lib2(x)$	Yes	Yes

(a) *client2*.

	Partition	Effect	Explored	Active
1	True	$RET = x$	Yes	No

(b) *lib2_old*.

	Partition	Effect	Explored	Active
1	$\neg(x > 10)$	$RET = x$	Yes	No
2	$x > 10$	$RET = 9$	No	No

(c) *lib2_new*.

Fig. 8. Paths explored for Example 2.

Proof Sketch. Suppose the equivalence assertion defined over complete active summaries is valid and the libraries are not client-specific equivalent. Using the result from Proposition 1, the validity of the equivalence assertion implies that there is a one-to-one mapping between the observable part of the active paths. Hence, there must exist an inactive path which cannot be matched. This contradicts with the fact that inactive path does not go through any change and therefore stays the same in both versions. The other direction is similar.

Freud+ Example. We now look at Freud+ on Example 2 from Fig. 2. In Example 2 the change is always active so we have to go through both client contexts (Rows 1-2 in Fig. 8a). When comparing active library paths which exercise changes, we opportunistically perform *concrete simulation* where a concrete input satisfying the current client context is used to replay on the paths from both library versions. In this example, we might use $x := 0$ as a concrete input which turns out not to be a real counterexample. After failing to quickly find a counterexample, we store the active path for later use and return to search for a new path in the client. In the case of *client2*, the next path that we find is shown in Row 2 of Fig. 8a. Having completed the exploration of the client, we generate an equivalence assertion involving only the active paths collected so far and try to prove its validity. This succeeds and thus the case for active CSE is identified for *client2*.

5 Evaluation

In this section, we describe our implementations of Freud and Freud+ and report on an empirical evaluation, aiming to answer the following two research questions: **RQ1**: how significant are the improvements in Freud+ compared to Freud? **RQ2**: how effective is our approach compared to the state-of-the-art?

5.1 Implementation

Our implementations[5] of Freud and Freud+ are built using PyExZ3 [2], a symbolic execution engine for Python written in Python, extended with four key modifications: summary generation, support for uninterpreted functions, exploration modulo calling context, and parallel exploration.

Summary generation is crucial to Freud's equivalence assertion generation. With this feature, PyExZ3 produces a partition-effect pair per path and maintains

[5] Available at: `https://github.com/FedericoAureliano/freud-tool`

a tree representation of the overall summary. *Support for uninterpreted functions* enables the top-down generation of summaries by allowing FREUD to explore the client irrespective of a particular version of its library. This feature is implemented as an extension of PyExZ3's interface with the Z3 [6] solver.

When FREUD+ encounters a call to an uninterpreted library, it uses the arguments and current path condition when exploring the two versions of the library. Our *exploration modulo calling context feature* collects and subsequently use this client context. Finally, the *parallel exploration* feature allows FREUD+ to explore both versions of a library while monitoring equivalence and finding potential counterexamples. This feature relies on summary generation for bookkeeping as the two versions of a function are explored.

The remaining aspect of our implementations is the driver which connects the above features, logs execution information, and provides an interface for running experiments. Our implementation changed 17 files in the PyExZ3 project,[6] adding 1,122 lines and deleting 443.

5.2 Method

We evaluated our approach on 39 benchmarks.[7] Each benchmark is either an abstracted version of a program found in the pre-study, an example we constructed ourselves, or the ModDiff benchmarks [22] which were originally written in C and translated, by us, to Python. We excluded 6 ModDiff benchmarks that did not consider updates to clients and use the remaining 23. Our experiments were performed on an Intel Core i5-6500 CPU with 15.6 GB of memory running Ubuntu 16.04. We did not limit the number of paths or the exploration depth of our symbolic execution engine. We set a timeout of 600 seconds for each benchmark.

We compared our tool with ModDiff which is the state-of-the-art in program semantic equivalence checking. A more detailed discussion of their approach can be found in Sec. 6. The running time of ModDiff are listed as reported by Trostanetski et al. [22]: we were unable to run ModDiff on our hardware or obtain the experimental set up used in [22] for replication.

5.3 Results

Fig. 9 shows the results for FREUD and FREUD+ over two metrics: the total number of paths explored, and execution times. The axes are logarithmically scaled. Each (blue) circle associated with the northern and eastern axes corresponds to the sum of paths explored for a particular benchmark. There are more circles in the south-east part of the figure because FREUD+ is often able to ignore paths in the libraries when proving equivalence and paths in the client when finding a counterexample. The only exceptions are cases where FREUD+ failed to run – at the time of writing, our implementation immediately rejects input programs with multiple calls to the library on the same path.

The (red) x's, associated with the southern and western axes, correspond to our tool's execution time. Again, the x's tend to be below the diagonal suggesting

[6] https://github.com/GroundPound/PyExZ3
[7] Available with implementation

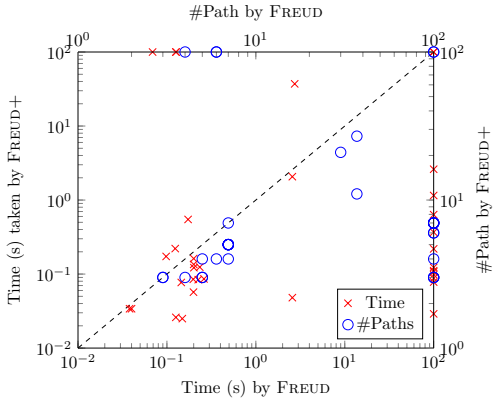

Fig. 9. Comparison of time and number of paths between Freud and Freud+.

Table 1. Execution time comparison of Freud, Freud+, and ModDiff.

Benchmarks	Times (s)		
	Freud	Freud+	ModDiff
add	0.038	0.034	0.2
const	0.040	0.034	0.541
loopmult2	–	0.111	1.689
loopmult5	–	0.368	3.88
loopmult10	–	0.635	9.543
loopmult15	–	2.609	21.55
loopmult20	–	1.153	49.031
loopunreach2	0.145	0.077	0.941
loopunreach5	0.200	0.135	1.126
loopunreach10	0.234	0.124	1.147
loopunreach15	0.200	0.162	1.191
loopunreach20	0.201	0.122	1.215
unchloop	–	0.219	2.838

(a) *CSE.*

Benchmarks	Times (s)		
	Freud	Freud+	ModDiff
loopmult2	–	0.029	3.451
loopmult5	–	0.098	5.914
loopmult10	–	0.107	10.614
loopmult15	–	0.088	14.024
loopmult20	–	0.123	25.795
loopunreach2	0.149	0.025	2.338
loopunreach5	0.198	0.085	3.216
loopunreach10	0.261	0.086	3.481
loopunreach15	0.199	0.057	3.446
loopunreach20	0.225	0.084	3.42

(b) *Not CSE.*

that Freud+ outperforms Freud. The x's on only the eastern axis, 12 total, represent benchmarks where Freud+ succeeded and Freud was unable to terminate. The single outlier x where both approaches terminated but Freud significantly outperformed Freud+ represents the benchmark labelled *is_prime3* – our implementation of Freud+ is not careful about repeating work and will re-summarize the libraries for each calling context. *is_prime3* exploits this deficiency. The success of Freud+ on both these measures supports a positive answer to **RQ1**: yes, our improvements are significant.

Table 1 compares our two implementations of Freud with ModDiff [22], over the ModDiff benchmarks. Table 2a lists the benchmarks that are semantically equivalent and the total execution times for each approach. "–" in the execution time column indicates that the tool did not terminate. Table 2a suggests that our technique is effective compared to the state-of-the-art, ModDiff, on deciding equivalence. Table 2b tells a similar story for non-equivalent cases. These results support a positive answer to **RQ2**: yes, our approach is effective and performs

well compared with the state-of-the-art. The complete results over all benchmarks and a detailed description for each benchmark can be found in Appendix B.

5.4 Threats to Validity

The applicability study in Appendix A a relies on manual determination of ground truth. To mitigate the risks of incorrect classification, we carefully documented every library and client decision. Two authors produced the documentation and made initial classifications; the remaining authors reviewed the results. The applicability study may not generalize due to our selection criteria and sample size. Further study is required to generalize the results.

The experiments in Sec. 5.2, are similarly vulnerable, and so the construction of the experimentation test suite followed a similar routine. Additionally, because our tool takes Python programs as input and ModDiff accepts C programs, we needed to translate ModDiff benchmarks to Python before performing the comparison. The choice of language undoubtedly affected the computation times so we ensured that our translations were not only semantically equivalent but also as similar as possible. To address the uncertainty associated with hardware mentioned in Sec. 5.2, we repeated our experiments on lower- and higher-powered machines. Our execution times remain consistent in both cases, with only slight variations.

6 Related Work

The approaches taken by Freud relate to several areas reviewed below.

Program Equivalence Checking. *Differential symbolic execution* (DSE) by Person et al. [18] is the closest work to ours. DSE performs standard symbolic execution on both program versions, before and after the change, and either reports that the two versions are equivalent or characterizes the *behavioral differences* by identifying the sets of inputs that cause different effects. It also introduced several notions of behavioral equivalence, including *partition-effect* and *functional equivalence*. However, these notions are defined over a whole program, without separation between clients and libraries.

Differential assertion checking (DAC) [16] defines program equivalence in terms of user-provided assertions. Given two program versions P and P' and a set of assertions s.t. all of them hold in P, DAC checks whether these assertions still hold in P'. Unlike in DSE, behavioral preservation does not have to be guaranteed across versions, and only a weaker form of equivalence with respect to the assertions is checked. We do not assume the availability of assertions, and rather generate equivalence assertions through client behavior exploration.

Regression verification [3,8] aims to formally prove that two program versions produce the same output for all inputs. Trostanetski et al. [22] recently proposed a *modular demand-driven* approach ModDiff to improve the scalability of such analysis. CSE, when configured with equivalence criteria based on functional equivalence, also falls into the regression verification framework. However, our problem is more specific in the sense that the change is restricted to the library program while the client is kept unchanged. In addition, there is no circular dependencies from the libraries to the client, and as such, we are able to opti-

mize equivalence checking by exploring the programs in a top-down way while significantly limiting the behaviors need to be considered.

Incremental Program Analysis. The work on *incremental verification* [21,5,7] aims to reuse results from prior verification as programs evolve, assuming that properties (of the client) to be verified are given. For example, Sery et al. [21] uses a compositional approach, implemented in a tool named eVolCheck, to summarize the properties of each procedure and then check whether these properties hold for the updated version of the program. Chaki et al. [5] uses state machine abstractions to analyze whether every behavior that should be preserved is still available (containment), and whether added behaviors conform to their respectful properties (compatibility). Fedyukovich et al. [7] offer an incremental verification technique for checking equivalence with respect to program properties. The primary focus of all these works is on reusing prior verification results as programs evolve while our work focuses on establishing equivalence w.r.t. a particular client. Furthermore, our work does not require specifications.

Symbolic Execution. Apart from DSE [18], there are a number of other symbolic execution-based approaches which are related to our work. *Directed incremental Symbolic Execution* (DiSE) [19] builds on DSE by adding static impact analysis for finding possible locations where the execution may vary. The insight of DiSE is to leverage the information extracted from the cheaper change impact analysis to enable more efficient symbolic execution of programs as they evolve. This is similar to our optimizations on collecting active paths which exercise changes, making subsequent analysis focused only on potentially changed behaviors.

Godefroid et al. introduced *demand-driven compositional symbolic execution* [1,12], the key novelty being compositionality: the search process is made compositional, and consequently exponentially faster than the non-compositional one [11]. Although these approaches do not address the problem of equivalence checking, our context-specific library exploration is inspired by them.

7 Conclusion and Future Work

In this paper, we defined the notion of Client-Specific Equivalence (CSE) and studied different equivalence criteria to parameterize it, namely functional and path equivalence. We presented a generic algorithm called FREUD, which relies on symbolic execution for checking CSE. We also introduced an extended version of the algorithm called FREUD+, which relies on the *incremental lazy exploration* strategy to improve the efficiency of FREUD. We implemented both algorithms and evaluated them on a set of benchmarks, confirming the applicability and efficiency of our approach.

As future work, we are interested in exploring other equivalence criteria, such as *partition-effect equivalence* [18] and in extending FREUD to handle more complex language constructs such as floats and heap manipulations. We are also interested in considering major library refactorings, including library splits and merges. Finally, proposing desirable fixes for the identified client-specific inequivalence will be another fruitful direction.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-Driven Compositional Symbolic Execution. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 367–381. Springer (2008)
2. Ball, T., Daniel, J.: Deconstructing Dynamic Symbolic Execution. In: Proc. of the 2014 Marktober Summer School on Dependable Software Systems Engineering. IOS Press (January 2015)
3. Benny Godlin, O.S.: Regression Verification: Proving the Equivalence of Similar Programs. J. Software Testing, Verification and Reliability 23(3), 241–258 (2013)
4. Cadar, C., Palikareva, H.: Shadow Symbolic Execution for Better Testing of Evolving Software. In: Proc. of ICSE NIER'14. pp. 432–435 (6 2014)
5. Chaki, S., Clarke, E., Sharygina, N., Sinha, N.: Verification of Evolving Software via Component Substitutability Analysis. Formal Methods in System Design 32(3), 235–266 (2008)
6. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proc. of TACAS'08. pp. 337–340. Springer (2008)
7. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Property-Directed Equivalence via Abstract Simulation. In: Proc. of CAV'16. pp. 433–453. Springer (2016)
8. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating Regression Verification. In: Proc. of ASE'14. pp. 349–360. ACM, New York, NY, USA (2014)
9. FLINT: Fast Library for Number Theory. `http://www.flintlib.org`
10. The GNU Multiple Precision Arithmetic Library. `https://gmplib.org`
11. Godefroid, P.: Compositional Dynamic Test Generation. In: Proc. of POPL'07. pp. 47–54. POPL '07, ACM, New York, NY, USA (2007)
12. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: Proc. of SAS'11. pp. 112–128. Springer-Verlag, Berlin, Heidelberg (2011)
13. Kawaguchi, M., Lahiri, S., Rebelo, H.: Conditional Equivalence. Tech. rep., Microsoft Research (October 2010)
14. King, J.C.: Symbolic Execution and Program Testing. Comm. of the ACM 19(7), 385–394 (1976)
15. Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do Developers Update Their Library Dependencies? Empirical Software Engineering pp. 1–34 (2017)
16. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential Assertion Checking. In: Proc. of ESEC/FSE'13 (2013)
17. The MPACK: Multiple Precision Arithmetic BLAS (MBLAS) and LAPACK (MLAPACK). `http://mplapack.sourceforge.net`
18. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential Symbolic Execution. In: Proc. of SIGSOFT FSE'08 (2008)
19. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed Incremental Symbolic Execution. In: Proc. of PLDI'11. pp. 504–515. ACM, New York, NY, USA (2011)
20. Rubin, J., Rinard, M.: The Challenges of Staying Together While Moving Fast: An Exploratory Study. In: Proc. of ICSE'16. pp. 982–993 (2016)
21. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental Upgrade Checking by Means of Interpolation-Based Function Summaries. In: Proc. of FMCAD'12. pp. 114–121. IEEE (2012)
22. Trostanetski, A., Grumberg, O., Kroening, D.: Modular Demand-Driven Analysis of Semantic Difference for Program Versions. In: Proc. of SAS'17. pp. 405–427. Springer (2017)

A A Pre-Study

Table 2. Results of the pre-study.

Project	Library	Clients	Affected	Unaffected
OpenSSL	*BN_is_prime_fasttest_ex*	10	5	5
OpenSSL	*RSA_check_key*	32	5	27
Linux	*gcd*	11	8	3
GMP	*mpf_get_d_2exp*	7	1	6

In this section, we evaluate the potential applicability of our approach via manual analysis of 60 clients calling four distinct libraries: *BN_is_prime_fasttest_ex (OpenSSL)*, *RSA_check_key (OpenSSL)*, *gcd (Linux)*, and *mpf_get_d_2exp (GMP)*. These libraries were selected because they come from popular open source projects written in C and are of a mathematical nature. OpenSSL and Linux have been starred on GitHub over 4,000 and 44,000 times respectively, while GMP, which is not hosted on GitHub, has been under active development for over 25 years.

Each library has a semantics-altering commit with the potential to affect its clients. The updates were selected because they make local changes that do not modify function signatures and are non-trivial.

Table 2 summarizes the results of our analysis, listing the overall number of clients of each library that we considered and the number of which have been affected or unaffected by the change. In the following, we provide a detailed description of the process we followed.

***BN_is_prime_fasttest_ex* (OpenSSL).** Library *BN_is_prime_fasttest_ex* receives four parameters: an integer a, an integer flag *do_trial_division*, and two structs used for call back procedure and context that are irrelevant to the change. The function aims to return 1 if a is prime, and 0 otherwise. *do_trial_division* specifies whether the function should attempt to divide a by a constant list of small primes. The change of interest[8] fixes a bug in which the original function considered small primes as composites because they are evenly divisible by a prime (themselves). After the commit, aptly titled "Small primes are primes too", the function checks that a candidate composite is not in the list of small primes. This change is irrelevant to clients that:

1. call *BN_is_prime_fastest_ex* with *do_trial_division = 0*;
2. call *BN_is_prime_fastest_ex* with a greater than the largest prime in the list of small primes; or
3. independently check if a is divisible by a small prime.

[8] Small primes are primes too (openssl/openssl@6e64c56): `https://github.com/openssl/openssl/commit/6e64c560663f5542fdc2580bb7b030c19b6919e4`

To find clients for this library, we searched GitHub using the string "BN_is_prime_fastest_ex", resulting in 11,500 C files. Of the first 1,000 code results[9], we found 10 unique clients (see Table 2). Five of them were unaffected by the change, calling *BN_is_prime_fastest_ex* with *do_trial_division = 0* (case #1). One of the remaining five[10] is "almost uneffected" as it calls *BN_is_prime_fastest_ex* with a large a that just falls short of the largest small prime. No client independently checks a over the list of small primes.

***RSA_check_key* (OpenSSL).** The *RSA_check_key* function takes in a pointer to a RSA key and decides its validity. RSA keys are composed of five integer fields: p, q, n, e, and d. The modification that we considered[11] adds a check that returns 0 (bad key) if any of these five components are null. This change is irrelevant to clients that:

1. call *RSA_check_key* with neither p, q, n, e, or d being null; or
2. will fail due to null value anyway.

To find clients for this library, we searched GitHub using the string "RSA_check_key". The first 1,000 search results out of the found 24,741 C files contained 32 unique clients (see Table 2). Of these, 27 were unaffected by this library change. 24 clients construct an RSA key by calling either *PEM_read_RSAPrivateKey*, *EVP_PKEY_get1_RSA*, or *RSA_generate_key* and then call *RSA_check_key* with this key. According to the documentation, these three helper functions successfully populate the RSA fields with non-null values or return null. The former situation corresponds to the first property of clients we are looking for, and the latter corresponds to the second (the library will fail given a null value before or after the update). In both situations, the change to the library is irrelevant to the client. There were clients[12] that attempted to access the fields before calling *RSA_check_key*. The change did not affect these clients because they will cause a segmentation fault before calling the library in the cases relevant to the change.

The five clients that are affected by this change receive the RSA key as an input parameter or use an unknown function to generate it (e.g., *parse_pk_file*[13]), and then call *RSA_check_key*.

***gcd* (Linux).** The Linux project's *gcd* function calculates the greatest common denominator of two unsigned integer values using the standard Euclidean algorithm. This function, in its original implementation, was vulnerable to division by

[9] GitHub only makes the first 1,000 results available.

[10] OpenSSL-Elgamal/elegamal_gen.c at 78039d35: `https://github.com/winscar/OpenSSL-Elgamal/blob/78039d3558bd05031003e0271b15273f4f9269e2/elgamal/elgamal_gen.c`

[11] Check for missing components in RSA_check (openssl/openssl@534e5fa): `https://github.com/openssl/openssl/commit/534e5fabadb18901daeff79f3a45-0d1b612880bd\#diff-b342dc84a6dfee097a144cda13e28e7a`

[12] docker-tor-hiddenservice-nginx/math.c at fbf15c7c: `https://github.com/opsxcq/docker-tor-hiddenservice-nginx/blob/fbf15c7ce4f5ed95158f92256333ca32d29d47bf/shallot/src/math.c`

[13] dudders/crypt_openssl.c at 63321d81: `https://github.com/p00ya/dudders/blob/63321d815e23ac896dcf4a9eafab3830b6b40cb6/crypt_openssl.c`

zero. To circumvent this issue, an update[14] was made to check that the smaller of the two input values is not zero. The change is irrelevant to clients that:

1. call *gcd* with non-zero values; or
2. are vulnerable to division-by-zero independently of this library.

The string "gcd" is too generic to be used for effectively searching GitHub. We thus limited our search to the Linux project itself, in which we found 11 clients. Of these, three are unaffected by the change. These clients either check that the inputs to *gcd* are non-zero directly (case #1), or use provably positive values (case #2). The remaining clients call the *gcd* function with values set by parameters. In such cases, we cannot be sure that the arguments to *gcd* are non-zero and thus conservatively classify these clients as affected (see Table 2).

***mpf_get_d_2exp* (GMP).** We also considered the function *mpf_get_d_2exp*, presented as a motivating example in Sec. 1. For a partial code listing, see Fig. 1a. This change affects the sign of the return when the input is negative, and is irrelevant to clients that:

1. do not use the returned double (i.e., only use the exponent);
2. call *mpf_get_d_2exp* with non-negative parameters; or
3. change the sign of the return of *mpf_get_d_2exp* when the input to this function is negative.

To find clients for this library, we searched GitHub using the string "mpf_get_d_2exp". This search yielded 7,632 C files. Of the first 1,000 code results, we found 7 unique clients, 6 of which were unaffected by the change (see Table 2). Three of the clients did not use the returned double (case #1), one always called the library with positive values (case #2), and one changed the sign of the return when necessary (case #3).

The one client affected by the change, shown in Fig. 1c, calls a function that is undefined on negative inputs with the result returned by *mpf_get_d_2exp*.

Summary. To summarize, our pre-study showed that in 68% of the cases, clients remain unaffected by the changes to the libraries. We believe that this is a representative figure because the library updates we considered were typical, and because clients were systematically selected without a bias. We thus conclude that the problem we are trying to address is of practical relevance.

B Experiment Details

In this section we describe our benchmarks and present the full results of our study in Sec. 5. For details on the ModDiff benchmarks see Trostanetski et al. [22]

We now give a description of the examples we created for our experiments. The complete results are given in Table 3. The function *BN_is_prime_fasttest_ex* (see Appendix A) inspired the first three programs: *is_prime1*, *is_prime2* and *is_prime3*. These programs use the same library but have different clients. The

[14] lib/gcd.c: prevent possible div by 0 (torvalds/linux@e968756): `https://github.com/torvalds/linux/commit/e96875677fb2b7cb739c5d7769824dff7260d31d`

library is a simple prime checker that takes an unsigned short x and a flag. If the flag is non-zero, the library returns 0; otherwise, it checks whether the input is prime by trial division over the first eight primes. The original version of the library does not check if a composite candidate is one of the first eight primes; the updated library does. The client for *is_prime1* always calls the library with the flag equal to 0; the client for *is_prime2* always calls the library with *x>19*; the client for *is_prime3* checks whether x is in the list of small primes before calling the library.

RSA_check_key and *gcd* (see Appendix A) inspired the next two programs, *divide1*, and *divide2*. The programs share a library that was patched to avoid a division by zero. *divide1*'s calls the library with safe arguments, while *divide2* is victim to the same divide-by-zero error as the library. Both programs are client-specific equivalent. In the latter case, the library update preserves partial functional equivalence since the original version is undefined on the value that differs.

mpf_get_d_2exp (see Appendix A) inspired *order*, *pos1*, and *pos2*. The library in *order* calculates an integer value whose sign depends on the order of its arguments. The update rectifies this by swapping values if necessary to ensure that the return is positive. The client for *order* is aware of this pitfall in the original library, and so it calls the library with arguments in the correct order to ensure a positive return. *pos1* and *pos2* share a client but have distinct library updates. In their case, the client always calls the library with a negative value. Since the libraries differ only on positive inputs, they are equivalent with regards to this client.

Of the remaining examples, *oneN1* is a case of a "strange" equivalence where the library changes significantly but the client remains unnafected. *oneN2*, and *get_sign* add to our collection of non-equivalent cases. *get_sign2, multiple*, and *ltfive* are straightforward equivalences. Finally, *odd* provides a case where an infinite loop in the library cannot be avoided but the client is unnafected.

Table 3. Complete results on evaluation of FREUD, FREUD+, and ModDiff. "N/A" indicates that a tool was not run on a particular benchmark. "–" indicates that the tool attempted the benchmark but either timed out or rejected it. The final three benchmarks contain a division by zero error. Both FREUD and FREUD+ found the error and reported it rather than deciding equivalence.

Benchmarks	Paths Explored					Times (s)			Comments
	Client	Lib1		Lib2					
		FREUD	FREUD+	FREUD	FREUD+	FREUD	FREUD+	ModDiff	
Equivalent									
get_sign2	2	3	3	2	2	0.173	0.547	N/A	
is_prime1	9	10	1	18	1	2.575	0.048	N/A	Inactive
is_prime3	9	10	9	18	9	2.735	37.146	N/A	Not checking for revisit
ltfive	2	2	1	2	1	0.124	0.221	N/A	
multiple	2	1	N/A	1	N/A	0.069	–	N/A	Lib(expr) in path
odd	2	–	–	1	1	–	–	N/A	Don't avoid ∞
oneN1	2	2	N/A	2	N/A	0.127	–	N/A	Two Lib
order	2	1	1	2	1	0.098	0.173	N/A	
pos1	2	–	–	2	2	–	–	N/A	Don't avoid ∞
pos2	2	–	1	2	1	–	0.078	N/A	Inactive
add	1	1	1	1	1	0.038	0.034	0.2	
const	1	1	1	1	1	0.040	0.034	0.541	
loopmult2	1	–	1	–	1	–	0.111	1.689	
loopmult5	3	–	1	–	2	–	0.368	3.88	
loopmult10	3	–	1	–	3	–	0.635	9.543	
loopmult15	3	–	1	–	3	–	2.609	21.55	
loopmult20	3	–	1	–	3	–	1.153	49.031	
loopunreach2	1	2	1	2	1	0.145	0.077	0.941	Inactive
loopunreach5	3	2	1	2	1	0.200	0.135	1.126	Inactive
loopunreach10	3	2	1	2	1	0.234	0.124	1.147	Inactive
loopunreach15	3	2	1	2	1	0.200	0.162	1.191	Inactive
loopunreach20	3	2	1	2	1	0.201	0.122	1.215	Inactive
unchloop	1	–	1	–	1	–	0.219	2.838	
Non-Equivalent									
get_sign	2/1	3	1	1	1	0.126	0.026	N/A	Exploration
is_prime2	2	10	9	18	10	2.574	2.069	NA	Assertion
oneN2	2	2	N/A	2	N/A	0.126	–	N/A	Two lib
loopmult2	1	–	1	–	1	–	0.029	3.451	Exploration
loopmult5	3	–	1	–	1	–	0.098	5.914	Exploration
loopmult10	3	–	1	–	1	–	0.107	10.641	Exploration
loopmult15	3/2	–	1	–	1	–	0.088	14.024	Exploration
loopmult20	3	–	1	–	1	–	0.123	25.795	Exploration
loopunreach2	1	2	1	2	1	0.149	0.025	2.338	Exploration
loopunreach5	3	2	1	2	1	0.198	0.085	3.216	Exploration
loopunreach10	3	2	1	2	1	0.261	0.086	3.481	Exploration
loopunreach15	3/2	2	1	2	1	0.199	0.057	3.446	Exploration
loopunreach20	3	2	1	2	1	0.225	0.084	3.42	Exploration
Other (Finds a bug)									
divide1						E	E	N/A	Div zero
divide2						E	E	N/A	Div zero
gcd						E	E	N/A	Div zero