Client-Specific Equivalence Checking

Federico Mora
Univ. of Toronto
Toronto, Canada
fmora@cs.toronto.edu

Yi Li
Univ. of Toronto
Toronto, Canada
liyi@cs.toronto.edu

Julia Rubin
Univ. of British Columbia
Vancouver, Canada
mjulia@ece.ubc.ca

Marsha Chechik
Univ. of Toronto
Toronto, Canada
chechik@cs.toronto.edu

ABSTRACT

Software is often built by integrating components created by different teams or even different organizations. With little understanding of changes in dependent components, it is challenging to maintain correctness and robustness of the entire system. In this paper, we investigate the effect of component changes on the behavior of their clients. We observe that changes in a component 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* (CSE) and develop an automated technique optimized for checking such equivalence. We evaluate our technique on a set of benchmarks, including those from the existing literature on equivalence checking, and show its applicability and effectiveness.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; *Dynamic analysis*;

KEYWORDS

Software change, equivalence checking, symbolic execution

ACM Reference Format:
Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-Specific Equivalence Checking. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238178

1 INTRODUCTION

Software systems are often composed of multiple related but independently developed components. Specifications of such components are usually limited to the description of their APIs. Yet, even upgrades that do not alter APIs can hinder the stability of dependent components [23]. Thus, dealing with component upgrades becomes a time-consuming task. This paper addresses the problem by investigating the impact of a change to a component (which we refer to as a "library") on its downstream consumers (which we refer to as "clients").

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09.
https://doi.org/10.1145/3238147.3238178

```
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.*

Figure 1: Two client-library pairs illustrating CSE.

Several existing techniques such as ModDiff [25], RVT [13], SymDiff [17], and Rêve [8] can be used for validating behavioral equivalence between two versions of a program or for identifying the precise set of changes between them. Yet, these equivalence checking techniques do not exploit the *usage pattern* of a particular library component within its client. We argue that the equivalence checking problem becomes more tractable when the usage pattern is considered. In particular, reasoning about a library update with respect to its impact on an (unmodified) client enables us to produce a highly efficient and accurate analysis.

Our preliminary analysis of 66 real-life clients of four large open-source libraries showed that, in 71% of the cases, the behavior of the client component is completely unaffected by a commit to one of its libraries (see Sec. 2 for more details). To illustrate, Fig. 1 shows two simplified client-library pairs, namely, *client1* which depends on *lib1* (Figs. 1a-c), and *client2* which depends on *lib2* (Figs. 1d-f). Each of the libraries has two versions, "*old*" and "*new*" For simplicity, we assume that all variables take arbitrary input values from the unbounded integer domain, so there are no overflows or underflows. For both client-library pairs, the changes in the libraries do affect how the libraries behave, but do not affect the clients' functional behaviors. For instance, even though *lib1_new* returns different values for inputs lesser than 10 compared to *lib1_old*, the behavior of *client1* remains exactly the same because of the way it calls the library. We say that the two versions of the libraries are equivalent with respect to this client.

In this paper, we provide a method for the effective checking of a special case of equivalence problems: whether two library components are equivalent with respect to a particular client. We formalize the notion of *Client-Specific Equivalence* (CSE) 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 with an effective implementation based on symbolic execution, Clever; and (3) an empirical evaluation of Clever versus state-of-the-art equivalence checking tools. Clever inherits the pitfalls of symbolic execution and the current implementation is limited to integer data-types.

The rest of the paper is structured as follows. Sec. 2 describes a preliminary study we conducted to assess the practical relevance of the CSE problem. Sec. 3 presents the overview of our approach on simple examples. Sec. 4 fixes notation and provides necessary background. We formally define our client-specific equivalence checking framework in Sec. 5. In Sec. 6, we describe our implementation and evaluate its effectiveness. Related work is discussed in Sec. 7. Sec. 8 outlines possible future directions and concludes.

2 APPLICABILITY STUDY

In this section, we assess the practical relevance of the CSE problem via manual analysis of 66 client-library function pairs. We found these pairs by searching GitHub for popular projects that provide functionality to other projects (libraries). We then identified signature preserving commits that modify the semantics of a library method and checked how these modifications affect the clients.

The library functions come from popular open source projects: Delorean, OpenSSL, Linux, and GMP. The first three projects have been "starred" on GitHub over 1,000, 4,000 and 44,000 times respectively, while GMP, which is not hosted on GitHub, has been under active development for over 25 years. Six of the client-library function pairs are written in Python, and the remaining 60 – in C. The automated analysis of the Python examples is discussed in Sec. 6.4. Here we briefly describe the libraries and clients.[1]

***Delorean.__init__* (Delorean).** The *Delorean.__init__* library function receives two parameters: *datetype* and *timezone* and returns a Delorean object that provides users with the datetime manipulation functionality. The behavior of the constructor is entirely defined by the type of arguments it receives. That is, all conditionals branch based on the type of *datetime* and *timezone*. There are two changes of interest for this library (`#679596a`, `#064bc8d`). Both changes, which concern the setting of an instance variable $\_tzinfo$ (timezone info), occur inside a check for *timezone* and *datetime* being *None*; the latter change occurs inside an additional check for *timezone* being of type *tzinfo*. We found three client functions and analyzed them with both library updates. All six pairs are unaffected by the library updates because they call the Delorean constructor with values of *datetime*, or *timezone* that avoid the change.

***BN_is_prime_fasttest_ex* (OpenSSL).** The 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 (`#6e64c560`) 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. We found 10 unique clients for this library: five of them call *BN_is_prime_fastest_ex* with *do_trial_division = 0*, avoiding the change.

***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 (`#534e5fa`) adds a check that returns 0 (bad key) if any of these five components are null. We found 27 clients that are unaffected by this library change. 24 of these 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. Additionally there are three clients (`#fbf15c7`) that attempted to access the fields before calling *RSA_check_key*. The change does not affect these clients because they will cause a segmentation fault before calling the library in the cases relevant to the change. We also found five clients that are affected by this change. These clients receive the RSA key as an input parameter or use an unknown function to generate it (e.g., *parse_pk_file*(dudders/crypt_openssl.c), 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. The original implementation of this function was vulnerable to division by zero. To circumvent this issue, an update (`#e968756`) was made to check that the smaller of the two input values is not zero. We found 11 clients for this library within the Linux project itself. Of these, three are unaffected by the change. These clients either check that the inputs to *gcd* are non-zero directly, or use provably strictly positive values. The remaining eight clients call the *gcd* function with values set by parameters and so may be affected by the change.

***mpf_get_d_2exp* (GMP).** We also considered the function *mpf_get_d_2exp*. For a partial code listing, see Fig. 2. This change affects the sign of the return when the input is negative. We found seven unique clients, six of which were unaffected by the change. Three of these six unaffected clients did not use the returned double, one always called the library with positive values, and one, shown in Fig. 3a, changed the sign of the return when necessary. The one client affected by the change, shown in Fig. 3b, calls a function that is undefined on negative inputs with the result returned by *mpf_get_d_2exp*.

Summary. Table 1 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 corresponding change. For 71% of the cases considered, clients remain unaffected by the changes to libraries. We thus conclude that the problem we are trying to address is of practical relevance.

[1] Full study details available at https://client-specific-equivalence-checker.github.io/

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

Figure 2: 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 }
```

(a) 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 }
```

(b) Client function *log_real* from the MPACK library [19].

Figure 3: An update and two sample clients of the *mpf_get_d_2exp* library function.

3 OVERVIEW

In this section, we give an overview of our approach for determining equivalences of libraries with respect to a particular client. We use the examples in Fig. 1 to illustrate the approach: these examples abstract the patterns observed in the applicability study of Sec. 2.

Example 3.1. Fig. 1a shows the source code of a client program; Figs. 1b and 1c 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. 1a). Thus, the change in *lib1_new* is never exercised. In fact, the two library versions are *conditionally equivalent* [15] 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. 1a), we say that that library versions are *client-specific equivalent* w.r.t. *client_1*.

Table 1: Results of the applicability study.

Project	Library Function	#Clients	#Affected	#Unaffected
Delorean	*Delorean()*	3	0	3
Delorean	*Delorean()[2]*	3	0	3
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

Example 3.2. Fig. 1d shows another client, *client2*. Figs. 1e and 1f show two versions of the library that *client* 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 a different client behavior: *lib2_old* always returns a value which is smaller than the input x, leading *client2* into the if-branch (Lines 2-3 in Fig. 1d). 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. 1d); yet, input value x is returned by *client2* in this case as well.

Unlike Example 3.1, here the change in the library is exercised by the client. Yet the two library versions are still client-specific equivalent: the change on the library is "digested" by the client so that the final output is unaffected. The example shown earlier in Fig. 3a falls into the same category.

Client-Specific Equivalence (CSE). In both examples, changes in the library programs do not affect the final return values of the client. We characterize client-specific equivalence in terms of the client's *observable behaviors*: we ignore the internal execution steps and only observe the input and output values of the client. This definition of equivalence is also referred to as *functional equivalence* [20].

Checking CSE. Now, we describe our general framework for determining client-specific equivalence. Fig. 4 overviews the architecture of the framework. The framework accepts as input a client and two versions of a library. We assume that library interfaces remain unchanged since the change can be easily caught by a compiler.

The framework's main components are a *behavior explorer* and an *equivalence verifier*. The behavior explorer analyzes behaviors of the libraries in terms of how they are used, and then it generates *equivalence assertions* that must be satisfied for the libraries to be CSE. The equivalence verifier checks the libraries against the equivalence assertions and either declares them to be equivalent

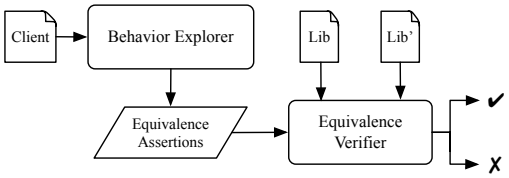

Figure 4: Architecture of the client-specific equivalence checking framework.

or provides a counter-example which demonstrates the behavioral differences observed in the client.

Incremental Lazy Exploration. Comparing Examples 3.1 and 3.2, we identify two types of CSE: (1) *inactive CSE* – that holds when changes are not exercised by the client and (2) *active CSE* – that holds when changes are anticipated, exercised, and specially handled by the client. The biggest distinction being 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. It helps to eliminate the main source of inefficiency of the generic CSE framework described above: fully exploring both the client and library programs before generating equivalence assertions based on the explored behaviors.

We call a path containing a call to the library an *active path*. Whenever an active path is explored, we effectively obtain a *client context* for the library call – a condition over client inputs under which the path is taken. With this context, we perform a *parallel exploration* of both library versions under the given context by shadowing [5] one with the other. This is done by simultaneously examining the same paths in both versions with symbolic execution [16]. 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.

In Example 3.1, we begin by exploring the client and skip the first path which does not involve library calls (row 1 in Fig. 5a). Then we explore the second path, shown in row 2, and perform a parallel exploration on both libraries with the client context $\neg(x > 10)$. The first rows of Figs. 5b and 5c 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. 6.

	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.*

Figure 5: Paths explored for Example 3.1.

4 BACKGROUND

In this section, we provide the necessary background on 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 function calls, and all variables range over integers. We assume that the type and number of input and output parameters are statically known for each function.

A program $P = (F_c, \{F_l\}_i)$ consists of a client and a set of library functions, such that the client calls the libraries. Each of the functions 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) calls to library functions, e.g., $\vec{x} \leftarrow F(\vec{y})$, where $\vec{x}$ and $\vec{y}$ are vectors of variables in $\mathcal{V}$ and F is a function 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 an infinite number of paths, such as in the case where there is a loop whose number of iterations depends on an unbounded variable.

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

For example, the function summary for *lib1_new* in Fig. 1c is $\varphi_{lib1'} = ITE(X > 10, RET = 9, RET = X)$, where ITE is the if-then-else operator, X is the input parameter, and RET is the output. $\varphi_{lib1'}$ is also complete since it covers both the case when $X > 10$ and when $X \leq 10$.

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

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

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

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

Figure 6: Semantics of operations.

Concrete, Abstract, and Observable Runs. We interpret semantics of 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. 6). 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 an *observable run* $\omega(\pi)$, hiding unobservable states and operations.

Recall *client1* shown in Fig. 1a. 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* (Lines 2 and 3), where $\top$ denotes an uninitialized value. By hiding the values of x, the observable part of the same runs can be written as simply $(2, \top) \rightarrow (3, X)$.

Symbolic Execution. *Symbolic execution* [16] 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.

5 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.

5.1 The CSE Checking Problem

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* only considering the input and output values of the client. $Behav(F)$ denotes the set of all (concrete) runs of a procedure F.

Definition 5.1 (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 $\{\omega(\pi) \mid \pi \in Behav(F_c[F_l])\}$ and $\{\omega(\pi') \mid \pi' \in Behav(F_c[F_l'])\}$ are equal.

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 their sets of concrete runs are equal. 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.

5.2 Checking Functional CSE

In most cases, there are 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 it is possible to reduce the CSE checking problem to the validity of first-order formulas. We begin by describing a general algorithmic framework – Clever,[2] 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, Clever explores 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 symbolic executing the procedure can be represented as a set of *partition-effect pairs* [20], 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 the summaries for two library versions produced in the previous step be $Behav(F_c[F_l])$ and $Behav(F_c[F_l'])$. Clever uses them to generate an *equivalence assertion* ϕ – a first-order formula with uninterpreted functions as placeholders for the libraries. The formula ϕ serves as a *mutual specification* [18] for the two libraries – the observable behaviors of the client are equivalent if and only if the library bodies respect

[2] Clever stands for CLient-specific EquiValence checkER.

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

$$\phi := \Big(\bigvee_{\hat{\pi}_i \in Behav(F_c[F_l])} (pc_i \wedge ob_i) \Leftrightarrow \bigvee_{\hat{\pi}'_i \in Behav(F_c[F'_l])} (pc'_i \wedge ob'_i) \Big),$$

where pc_i and ob_i are path and effect constraints 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 to constrain the uninterpreted function placeholders in the equivalence assertion ϕ. Finally, we check the validity of ϕ composed with the library summaries. If a violation is found, we report a counterexample; otherwise, if the generated summaries are complete, we proved functional CSE.

Proposition 5.2. *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. 5.1, F_l and F'_l are CSE since the equality between concrete runs is established by the logical equivalence between composed summaries.

5.3 Incremental Lazy Exploration

It may not always be possible to compute complete summaries of client and libraries due to limitations of symbolic execution techniques. We now present an optimized behavior exploration strategy, *incremental lazy exploration*, which prioritizes establishing proof arguments for *inactive CSE* (see Sec. 3) and disproving CSE through early discovery of counterexamples. This allows us to speed up the CSE checking process and obtain partial results even if the path exploration of some functions is not exhaustive.

A pseudo-code implementation of Clever with incremental lazy exploration is given in Fig. 7 and the workflow for this technique – in Fig. 8. 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 interested in exploring and reasoning about active paths. We call a summary consisting of only active paths an *active summary*.

The while loop on 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, resulting in $Behav_p(F_l)$ and $Behav_p(F'_l)$, respectively. If these summaries exercise the difference in the library versions, checked on Line 6, then Clever does two things. First, it finds a concrete value that satisfies p and checks whether it is a counterexample. If affirmative, Clever reports that F_l and F_l' are not client-specific equivalent. Second, Clever updates the active summaries. If no active paths have been found after the client has been fully explored, Clever reports equivalence. Otherwise, Clever uses the active summaries to generate an assertion (Line 14), and then check it (Line 15).

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** Clever(F_c, F_l, F'_l)
2: $A \leftarrow Initial$ ▷ Initialize active summaries of F_c, F_l, F_l'
3: **while** $p \in$ Explore($F_c[f_l(\vec{x})]$) **do**
4: $Behav_p(F_l) \leftarrow$ Explore($F_l(\vec{x}) \mid \vec{x} \models p$)
5: $Behav_p(F'_l) \leftarrow$ Explore($F'_l(\vec{x}) \mid \vec{x} \models p$) ▷ Summaries mod p
6: **if** p uses change **then**
7: **if** concrete value for p is a counterexample **then**
8: **return** false
9: **end if**
10: A.update($p, Behav_p(F_l), Behav_p(F'_l)$)
11: **end if**
12: **end while**
13: **if** Empty(A) **then return** true
14: $\phi \leftarrow$ AssertGen(A)
15: **if** Verify(ϕ, F_l, F'_l) **then return** true
16: **else return** false
17: **end procedure**

Figure 7: Algorithm for checking CSE based on symbolic execution and lazy path exploration.

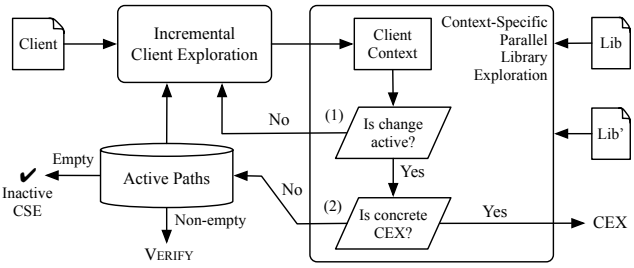

Figure 8: Clever workflow with incremental lazy exploration.

Proposition 5.3. *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.*

Proof Sketch. Suppose the equivalence assertion defined over complete active summaries is valid and the libraries are not client-specific equivalent. By Proposition 5.2, the validity of the equivalence assertion implies that the observable part of the active paths are equal. Hence, there must exist an inactive path which cannot be matched. This contradicts the fact that inactive path does not go through any change and therefore stays the same in both versions. The other direction is similar.

Finally, if we fail to discover counterexamples or fully explore all client paths with limited resources, we are still able to partially prove or disprove equivalence for the considered client contexts. A different equivalence checking technique can be used in this case to decide equivalence of the remaining explored contexts.

Clever Example. We now illustrate Clever using Example 3.2 from Fig. 1. In Example 3.2, the change is always active so we have to go through both client contexts (Rows 1-2 in Fig. 9a). When comparing active library paths which exercise changes, we opportunistically perform *concrete simulation* where a concrete input

	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*.

Figure 9: Paths explored for Example 3.2.

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. 9a. 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*.

6 EVALUATION

In this section, we describe our implementation of Clever, report on an empirical evaluation, and present a case-study. We aim to answer the following research questions: **RQ1**: how effective is Clever compared to the state-of-the-art equivalence checking techniques for checking CSE? **RQ2**: how significant is the impact of incremental lazy exploration on effectiveness of Clever? **RQ3**: can Clever be effectively applied to real software projects?

6.1 Implementation

Our implementation[3] of Clever is built on top of PyExZ3 [2], a symbolic execution engine for Python written in Python, and PySMT [10], a Python interface to SMT solvers. As a consequence of this combination, our tool is currently limited to integer reasoning. The four key components of our software are summary generation, support for uninterpreted functions, exploration modulo calling context, and parallel exploration.

Summary generation is crucial to Clever's equivalence assertion generation. This feature symbolically executes programs returning a set of partition-effect pairs, one for each explored symbolic path. Support for *uninterpreted functions* enables the top-down generation of summaries by allowing client exploration irrespective of a particular version of its library.

When Clever encounters a call to a library, it uses the arguments and current path condition as a context when exploring the two versions of the library. Our *exploration modulo calling context* feature collects and subsequently uses this client context. Finally, the *parallel exploration* feature allows Clever to explore both versions of the library while monitoring equivalence and discovering potential counterexamples early. The feature is based on shadow symbolic execution [5] but incorporates summary generation directly.

Finally, we implemented a driver which connects the above features, logs execution information, and provides an interface for running experiments. Our implementation changed 26 files in the PyExZ3 project, modifying approximately 1,000 lines of code.

PyExZ3 was not the only choice of symbolic execution engine, but its focus on extensibility and its combination of existing features made it a natural choice. We also considered KLEE [4] which can process C programs and implements shadow symbolic execution [5]. However, KLEE does not support uninterpreted functions or summary generation and was not as amenable to the necessary modifications as PyExZ3.

[3]Code is available at https://client-specific-equivalence-checker.github.io/

6.2 Comparison with Existing Tools

In this section, we report the results of comparing Clever with well-established equivalence checking tools.

Existing Tools. We compared Clever, with four state-of-the-art tools in equivalence checking and regression verification: RVT [13], SymDiff [17], ModDiff [25], and Rêve [8]. All of these tools can be used to prove *partial equivalence* of two related programs, i.e., producing the same outputs for all possible inputs given that both programs terminate. Therefore, technically, they can also be used to check functional CSE by considering the client-library pair as a whole. However, since they are not designed for checking CSE specifically, they would not be able to leverage the fact that the clients are unmodified and the particular usage patterns observed in the clients. A more detailed discussion of these approaches is found in Sec. 7.

Subjects and Methods. We evaluated all tools on 60 benchmarks, with each benchmark consisting of a pair of programs before and after some changes in the library. We used 23 benchmarks from the ModDiff suite [25] (in C). We excluded six ModDiff benchmarks that do not perform updates to libraries. Some of the ModDiff benchmarks are indexed, e.g. *loopmult5* is related to *loopmult10*. We augmented the suite with 16 additional benchmarks that continue these indexed patterns. 21 benchmarks are examples extracted from programs that we inspected during the applicability study (in C or Python) and examples we constructed ourselves (in Python). In total, 29 pairs were equivalent and 21 – inequivalent.

To run the different tools on the benchmark, we translated all of the programs to C (required by RVT, ModDiff, Rêve), and Python (required by Clever) manually; and to Boogie [3] (required by SymDiff) with SMACK [22].

We performed the experiments on an Intel Core i7 4.00 GHz CPU with 16 GB of memory running Windows 10 with Cygwin. For each benchmark, we set a timeout of 300 seconds.

Results. Table 2 shows the results of comparing Clever with the other four state-of-the-art equivalence checking tools. The execution times are measured in seconds and the winner for each benchmark is in boldface. Clever solves the biggest number of benchmarks and outperforms all of the other tools in the majority of the cases. For Rêve, many instances that time out on our machine, using the desktop distribution, terminate in seconds when run using

Table 2: Run-time in seconds of Clever, ModDiff, Rêve, RVT, and SymDiff, where "–" indicates that the tool either times out or reports inconclusive results and "×" indicates an error.

	Benchmarks	Clever	ModDiff	Rêve	RVT	SymDiff
Equivalent	divide	**0.089**	×	–	5.870	–
	factorial	**0.295**	–	–	–	–
	fib	**0.268**	×	–	–	–
	get_sign2	0.068	**0.032**	–	–	–
	is_prime1	**0.056**	2.980	–	–	–
	is_prime3	**1.289**	7.910	–	–	–
	ltfive	**0.108**	0.792	–	–	–
	multiple	**0.077**	0.070	–	–	–
	order	**0.035**	0.122	–	5.000	–
	pos2	–	–	–	–	×
	pos3	**0.085**	–	–	–	–
	oneN2	0.107	**0.048**	–	–	–
	oneBound	0.089	**0.008**	–	4.800	–
	add	0.063	**0.010**	< 1	4.257	8.480
	const	0.062	**0.024**	< 1	4.118	8.760
	loopunreach2	0.061	**0.051**	< 1	–	–
	loopunreach5	0.104	**0.068**	< 1	–	–
	loopunreach10	0.104	**0.067**	< 1	–	–
	loopunreach15	0.110	**0.068**	< 1	–	–
	loopunreach20	0.105	**0.067**	< 1	–	–
	unchloop	**0.065**	0.141	< 1	–	–
Non-Equivalent	divide2	**0.100**	×	< 5		
	factorial	**0.081**	0.089	–		
	fib	**0.216**	–	–		
	get_sign	0.066	**0.047**	< 5		
	is_prime2	**0.116**	9.790	< 5		
	loopunreach2	**0.062**	0.117	< 1		
	loopunreach5	**0.078**	0.154	< 1		
	loopunreach10	**0.078**	0.156	< 1		
	loopunreach15	**0.079**	0.156	< 1		
	loopunreach20	**0.075**	0.143	< 1		
	odd	–	–	–		
	pos	–	–	–		
	oneN1	**0.075**	×	< 5		

the online version of the tool. To optimally capture Rêve's capabilities, we report categorical results collected from running the online version: termination in < 1, and < 5 seconds, and non-termination in over 300 seconds or returning unknown (–).

Since RVT and SymDiff are not designed for disproving equivalence, we did not include them in the comparison for non-equivalent cases. Similar to the experiment results reported in [25], RVT and SymDiff only solved four and two benchmarks, respectively. SymDiff crashed with an unhandled exception for *pos2* while ModDiff reported an incorrect result for *divide*, *fib*, *divide2*, and *oneN1* (all marked by "×"). Rêve performs well on non-equivalent cases and struggles on proving equivalence for benchmarks which require non-trivial relational invariants, e.g., *divide*, *factorial*, *ltfive*, etc. There are also a few cases, including *pos2*, *odd*, and *pos*, that no tool could solve within the given time limit. The reason was that they all have an unbounded loop with a non-trivial loop condition.

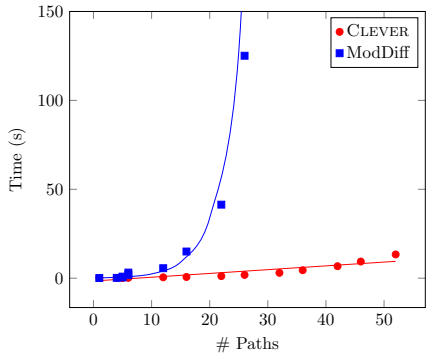

Figure 10: Clever and ModDiff execution times versus number of paths over fifteen client-specific equivalent problem instances.

Looking at the equivalent cases, Clever performed significantly better than the rest of the tools on *divide*, *is_prime1*, *pos3*, *loopmult15* and *loopmult20*, which demonstrate the benefits of early detection of inactive CSE. A common trait of these benchmarks is that the analysis of a long-running loop in the library could be avoided if the *top-down* analysis considered the specific range of input values coming from the client. For instance, both of the library functions in the *loopmult* examples both compute the product of two input parameters, but in slightly different ways. In oder to prove that they are equivalent, a bottom-up analysis of RVT and SymDiff would need to establish a non-linear relational invariant between the library functions, namely, $(RET == x \times y) \wedge (RET' == y' \times x')$, where x, y, x', and y' are the input parameters. Yet Clever only needs to consider the few cases defined by the particular inputs provided by the clients.

For the non-equivalent cases, Clever was almost always the best among the four. This shows the advantages of the parallel exploration feature which disproves equivalence by discovering counterexamples early on.

Overall, Clever and ModDiff were comparable in most of the cases, and the differences in their run-times were often under a second. In all of the cases where this difference was larger, the performance of Clever was superior. This benefit is illustrated in Fig. 10 which plots the execution times of Clever and ModDiff on the set of equivalent *loopmult* instances. The first five instances, *loopmult{2,5,10,15,20}*, were directly taken from the ModDiff suite. We extended the pattern to analyze the growth of the execution times. Fig. 10 shows results consistent with Table 2. When the instance in question is simple (as measured in the number of paths), the two tools perform similarly. As the difficulty increases, ModDiff's execution time goes up significantly while Clever's grows linearly. The non-equivalent *loopmult* instances show a similar but more pronounced difference. This is because Clever is able to ignore most library paths due to the client usage of the library.

Answer to RQ1: Yes, the comparison results suggest that our technique is effective and efficient on deciding client-specific equivalence, compared with the state-of-the-art.

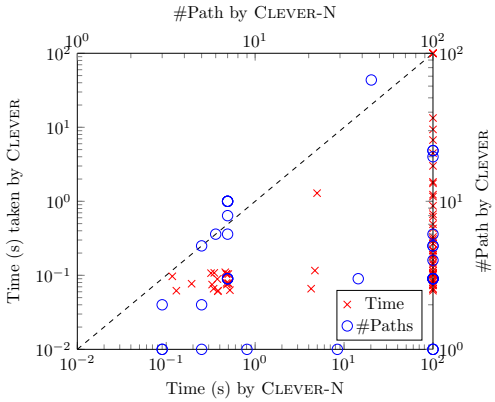

Figure 11: Comparison of time and number of paths between Clever-N and Clever.

6.3 Effect of Incremental Lazy Exploration

To evaluate the effect of incremental lazy exploration, we created a version of our tool without this feature, named Clever-N, and compared it to Clever on the set of benchmarks described in Sec. 6.2.

Fig. 11 shows the results over two metrics: the total number of paths explored, and execution times. The axes are logarithmically scaled. Each (blue) circle shows the comparison of the two tools in terms of the number of paths explored for a particular benchmark – the top and right axes represent the results by Clever-N and Clever, respectively. There are more circles below the diagonal line because Clever is often able to ignore paths in the libraries when proving equivalence and paths in the client when finding a counterexample. There are some circles above the diagonal line because two different calling contexts can lead to the exploration of the same library path, and because Clever does not avoid exploring the same path multiple times (at the time of writing).

Similarly, each (red) cross shows the comparison of Clever-N and Clever in terms of their execution time. The bottom and left axes represent the running time of Clever-N and Clever, respectively. Again, the crosses tend to be below the diagonal, suggesting that Clever outperforms Clever-N in most of the cases. The crosses lying on the right axis represent benchmarks where Clever succeeded and Clever-N was unable to terminate. These include most of the *loopMult* examples coming from the ModDiff benchmarks. Exploiting the specific client contexts, Clever avoids exploring the exponential number of paths for the while loops in the libraries and thus saves the execution time.

> **Answer to RQ2:** Incremental lazy exploration has a positive and pronounced impact on the effectiveness of Clever both in terms of running time and the number of paths explored.

6.4 Case Study

In Sec. 2, we described six Python client-library function pairs. In this section, we report on the experience applying Clever to these pairs, aiming to answer **RQ3**.

```
+ SECOND = 1
+ MINUTE = 2
def truncate(self, s):
-   if s == 'second':
+   if s == SECOND:
        self._dt = self._dt.replace(microsecond=0)
-   elif s == 'minute':
+   elif s == MINUTE:
    ...
```

Figure 12: Example string to integer transformation for *Delorean.truncate*.

Preparation. To prepare the examples as a valid input to Clever, we (Step 1) made the necessary simplifications to external functions; (Step 2) transformed string operations to integer operations; and (Step 3) wrapped the client function with an entry point for symbolic execution. We describe these in turn below.

Step 1. To illustrate the required simplifications to external functions, consider the Python datetime function *now()*. It is intended to return the time at invocation, but to avoid spurious counterexamples to client-specific equivalence, we simplified it to return a fixed time.

Step 2. We illustrate a transformation from string operations to integer operations in Fig. 12. This figure shows a snippet of a Delorean method that reasons by cases on a single argument, and the corresponding transformation. Because the original reasoning is limited to checking equality against constant string values, we transform the function to check integer constants, ensuring that the mapping from string to integer constants is a bijection, the empty string is mapped to 0, inputs are transformed accordingly, and the changes are propagated to all dependent code.

Step 3. We wrap client functions with an entry point for symbolic execution so that all arguments and function returns are integers. The wrapping function constructs the necessary objects to pass into the client function, and then deconstructs the output.

Results. Since all six client-library method pairs are client-specific equivalent (see Sec. 2), we added 10 modified examples intended to break the equivalence. Clever correctly classified all 16 examples, with an average execution time of 1.517 seconds.[4]

> **Answer to RQ3:** Yes, our approach can be effectively applied to real software projects. However, our current implementation requires some manual program preprocessing.

6.5 Threats to Validity

The applicability study in Sec. 2 relies on manual determination of ground truth. To ensure internal validity, we had two authors independently classify each client-library function pair, and a third author settle conflicting cases.

We maintained three versions of the benchmarks used in Sec. 6.2 (Python, C, Boogie). To enable cross-language comparisons between the tools, we limited the examples to subsets of the languages that can be trivially translated: control flow, integer operations, and no

[4] All examples are available at https://client-specific-equivalence-checker.github.io/

heap manipulations. Furthermore, since all the tools assume unbounded integers, C's *int* data type corresponds directly to Python's for our comparisons. The conversion between C and Boogie was done automatically, using SMACK [22].

Our encouraging results presented in the applicability study in Sec. 2, in the comparisons in Sec. 6.2, and in the case-study in Sec. 2 may not generalize to all software projects. Yet we believe that our selection process of the sample populations used in these sections resulted in software representative of current best practice.

7 RELATED WORK

Clever relates to several techniques reviewed below.

Program Equivalence Checking. *Differential symbolic execution* (DSE) by Person et al. [20] 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. SymDiff [17] also checks for partial equivalence between pairs of procedures in a program, and its notion of *mutual summary* [14] can also be used to encode client-specific equivalence. For example, SymDiff implements *differential assertion checking* (DAC) [18], which 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 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. In principle, CSE can be phrased as DAC by asserting that the input-output relations of the client stay the same even when the library changes. However, SymDiff can only prove equivalences, and not disprove them [25]. In addition, as shown by our experiments (see Sec. 6.2), SymDiff often fails to automatically infer mutual summaries which are strong enough to conclude functional equivalence.

Regression verification [8, 13] aims to formally prove that two programs are functionally equivalent. RVT [13] proves *partial equivalence* of two related programs, i.e., that they produce same outputs for all inputs given that both of them terminate, according to a set of proof rules. Recursive calls are first abstracted as uninterpreted functions, and then the proof rules for non-recursive functions are discharged in a bottom-up fashion, which makes it difficult to exploit specific calling contexts for specific clients. Rêve [8] targets programs with complex arithmetic and control flows, and automatically proves equivalence when "simple" coupling predicates over linear arithmetic exist for the two programs. Trostanetski et al. [25] recently proposed a *modular demand-driven* approach, ModDiff, to improve the precision and scalability of such analysis. When considering the client and library as a whole, CSE also falls into the regression verification framework. However, our problem is more specific in the sense that the change is restricted to the library part while the client is kept unchanged. In addition, there are no circular dependencies from the libraries to the client, allowing us to optimize equivalence checking by exploring the programs top-down while significantly limiting the behaviors that need to be considered.

Incremental Program Analysis. The work on *incremental verification* [6, 7, 24] 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. [24] 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. [6] 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 w.r.t. program properties. The primary focus of all these works is 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 [20], there are a number of other symbolic execution-based approaches which are related to our work. *Directed incremental Symbolic Execution* (DiSE) [21] 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 of 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.

8 CONCLUSION AND FUTURE WORK

In this paper, we defined the notion of Client-Specific Equivalence (CSE) and presented an algorithm called Clever, which leverages heuristics tailored for checking CSE. We implemented a prototype for Clever and compared it with four state-of-the-art equivalence checking tools on a set of non-trivial benchmarks. We also evaluated our approach on a real-world case study, confirming its applicability and efficiency.

As future work, we intend to apply Clever to more diverse systems, extending it to support other programming languages and language constructs such as heap manipulations. We also intend to add support for floating point numbers, strings, and objects composed of these primitives. Beyond that, we are interested in exploring other definitions of equivalence, such as *path equivalence* and *partition-effect equivalence* [20], which give stronger guarantees but might be more expensive to check. Finally, proposing desirable fixes for the identified client-specific inequivalence is another fruitful direction.

REFERENCES

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Proc. of TACAS'08 (LNCS)*, Vol. 4963. Springer, 367–381.

[2] Tom Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Proc. of the 2014 Marktober Summer School on Dependable Software Systems Engineering*. IOS Press.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. of FMCO'05*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, 364–387.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tesets for Complex Systems Programs. In *Proc. of OSDI'08*.

[5] Cristian Cadar and Hristina Palikareva. 2014. Shadow Symbolic Execution for Better Testing of Evolving Software. In *Proc. of ICSE NIER'14*. 432–435.

[6] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. 2008. Verification of Evolving Software via Component Substitutability Analysis. *Formal Methods in System Design* 32, 3 (2008), 235–266.

[7] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property-Directed Equivalence via Abstract Simulation. In *Proc. of CAV'16*. Springer, 433–453.

[8] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proc. of ASE'14*. ACM, 349–360.

[9] Flint 2017. FLINT: Fast Library for Number Theory. http://www.flintlib.org.

[10] Marco Gario and Andrea Micheli. 2015. pySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms. In *SMT Workshop*.

[11] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proc. of POPL'07*. ACM, 47–54.

[12] Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. 2011. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proc. of SAS'11*. Springer-Verlag, 112–128.

[13] Benny Godlin and Ofer Strichman. 2013. Regression Verification: Proving the Equivalence of Similar Programs. *J. Software Testing, Verification and Reliability* 23, 3 (2013), 241–258.

[14] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In *Proc. of CADE'13*. Springer-Verlag, 282–299.

[15] Ming Kawaguchi, Shuvendu Lahiri, and Henrique Rebelo. 2010. *Conditional Equivalence*. Technical Report. Microsoft Research.

[16] James C. King. 1976. Symbolic Execution and Program Testing. *Comm. of the ACM* 19, 7 (1976), 385–394.

[17] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proc. of CAV'12*. Springer-Verlag, 712–717.

[18] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proc. of ESEC/FSE'13*.

[19] MPACK 2017. The MPACK: Multiple Precision Arithmetic BLAS (MBLAS) and LAPACK (MLAPACK). http://mplapack.sourceforge.net.

[20] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential Symbolic Execution. In *Proc. of SIGSOFT FSE'08*.

[21] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *Proc. of PLDI'11*. ACM, 504–515.

[22] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations: Decoupling Source Language Details from Verifier Implementations. In *Proc. of CAV'14*, Vol. 8559. Springer, 106–113.

[23] Julia Rubin and Martin Rinard. 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *Proc. of ICSE'16*. 982–993.

[24] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012. Incremental Upgrade Checking by Means of Interpolation-Based Function Summaries. In *Proc. of FMCAD'12*. IEEE, 114–121.

[25] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-Driven Analysis of Semantic Difference for Program Versions. In *Proc. of SAS'17*. Springer, 405–427.