Scaling Client-Specific Equivalence Checking via Impact Boundary Search

Nick Feng
fengnick@cs.toronto.edu
University of Toronto

Federico Mora
fmora@cs.berkeley.edu
University of Calfornia, Berkeley

Vincent Hui
vhui@cs.toronto.edu
University of Toronto

Marsha Chechik
chechik@cs.toronto.edu
University of Toronto

ABSTRACT

Client-specific equivalence checking (CSEC) is a technique proposed previously to perform impact analysis of changes to downstream components (libraries) from the perspective of an unchanged system (client). Existing analysis techniques, whether general (regression verification, equivalence checking) or special-purpose, when applied to CSEC, either require users to provide specifications, or do not scale. We propose a novel solution to the CSEC problem, called 2clever, that is based on searching the control-flow of a program for *impact boundaries*. We evaluate a prototype implementation of 2clever on a comprehensive set of benchmarks and conclude that our prototype performs well compared to the state-of-the-art.

ACM Reference Format:
Nick Feng, Federico Mora, Vincent Hui, and Marsha Chechik. 2020. Scaling Client-Specific Equivalence Checking via Impact Boundary Search. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416634

1 INTRODUCTION

Software systems are often composed of multiple independently developed but related components. Upgrades to these components, even those that do not alter APIs, can hinder the stability of the system [31], making component upgrades a complex and time-consuming task. Several existing techniques, such as ModDiff [34], RVT [17], SymDiff [24], and Rêve [15], 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 techniques do not exploit the *usage pattern* of a particular library component within its client.

In our earlier work [27], we argued that the equivalence checking problem becomes more tractable when the usage pattern is considered. We further defined the *client-specific equivalence checking (CSEC)* problem as that of determining the impact of changes to downstream components (libraries) from the perspective of an unchanged system (client). We also argued for the practical relevance of CSEC in an applicability study, and proposed a solution to CSEC in a symbolic execution-based tool called Clever.

While Clever performs better than the general-purpose differential program analysis techniques on instances of the CSEC problem, it is still unable to handle realistic programs. Specifically, Clever struggles in cases of complex clients, multiple library calls, and cases where a finite set of paths is insufficient to solve the CSEC problem. Existing techniques share this scalability issue because they produce monolithic queries that their reasoning engine can't handle, e.g., Rêve; they are path-based, e.g., ModDiff; or they depend on expensive invariant inference techniques, e.g., SymDiff. Instead of generating one difficult query to an underlying reasoning engine, we propose an approach that generates a sequence of smaller, simpler queries. These smaller queries frequently have a finite-number of paths, can be handled by invariant inference techniques, and imply overall equivalence.

Illustrative Example. Consider the client and two libraries in Fig. 1. Fig. 1a shows the client, `sum_primes`, which takes an integer `x` and returns the sum of all primes between `0` and `x`. `sum_primes` depends on a library, `composite`, to check if numbers are prime. Figs. 1b and 1c show two versions of `composite`. The first library version, `composite_0`, returns `0` if its input is prime, and the number of factors of the input otherwise. The second, `composite_1`, returns `0` if its input is prime, and `1` otherwise.

Existing techniques—in particular, Clever, ModDiff, SymDiff, RVT, and Rêve—fail to prove that `composite_0` and `composite_1` are client-specific equivalent for `sum_primes`. Clever and ModDiff struggle because the problem has an infinite number of paths; SymDiff and Rêve's reasoning engines are unable to identify a relational invariant that is strong enough to prove equivalence; and RVT's bottom-up approach fails to use the crucial client context.

In this paper, we propose a novel approach, 2clever, that scales better than existing tools. 2clever hinges on two key observations: (1) in practice, a small portion of client control-flow graph (CFG) is often sufficient to prove that a library call does not affect the client—we call such a sub-CFG *an impact boundary*; and (2) if all library calls have an impact boundary, then the library update does not affect the client.

For example, 2clever determines that `sum_primes` in Fig. 1a is unaffected by the change to `composite` in Figs. 1b and 1c by bounding the impact of the library call in `sum_primes`. More specifically, 2clever proves that the impact is contained within the loop, i.e., that there does not exist a value for `x`, `j`, and `sum` such that a single pass of

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416634

```c
int sum_primes(int x){
  int sum = 0;
  int j = 0;
  while (j < x){
    if (composite(j) == 0)
      sum = sum + j;
    j++;}
  return sum}
```

(a) `sum_primes` in C.

```c
int composite_0(int num) {
  int count = 0;
  int i = 2;
  while(i < num) {
    if(num%i==0)
      count++;
    i++;}
  return count;}
```

(b) `composite_0` in C.

```c
int composite_1(int num) {

  int i = 2;
  while(i < num) {
    if(num%i==0)
      return 1;
    i++;}
  return 0;}
```

(c) `composite_1` in C.

Figure 1: The client, `sum_primes`, returns the sum of all primes less than `x`. The client calls `composite` to check for primality. The original version, `composite_0`, returns the number of factors of its input; the new verison, `composite_1`, returns `1` if it finds a factor, and `0` otherwise. The client is unaffected by their difference.

the loop using `composite_0` produces different values for `x`, `j`, and `sum` at the end of the loop compared to using `composite_1`. Since no library call affects the client, 2clever concludes that `composite_0` and `composite_1` are client-specific equivalent. Using the same strategy, 2clever is able to efficiently solve many other instances that other existing tools are unable to handle. Furthermore, 2clever can be implemented as a strategy on top of existing techniques with relatively low overhead. This means that, in general, 2clever can handle any instance that existing tools can handle.

Contributions. This paper makes the following contributions. (1) We observe that the CSEC problem can usually be solved using limited portions of the client CFG. These portions are *extractable* sub-CFGs that imply equivalence; we call them *impact boundaries*. (2) We develop an approach, 2clever, that searches for impact boundaries. It makes a novel connection between the classic literature on procedure extraction and equivalence checking. (3) We describe an efficient realization of 2clever with two important components: an algorithm for finding extractable sub-CFGs and a bespoke equivalence checking algorithm. (4) We report on a prototype implementation of 2clever and empirically evaluate it on a suite of 568 benchmarks taken or constructed from related work.

Organization. The rest of this paper is organized as follows. Sec. 2 gives the necessary formal background. Sec. 3 describes our approach at a high level, including the definition of impact boundaries and a naive impact boundary search strategy. Sec. 4 describes an advanced impact boundary search. Sec. 5 reports on the implementation of these ideas. Sec. 6 evaluates the performance of 2clever compared to state-of-the-art techniques. Sec. 7 surveys related approaches. We conclude in Sec. 8 with the summary of the paper and discussion of future research directions.

2 FORMAL BACKGROUND

This section describes control-flow automaton (CFA), formally defines the CSEC problem in terms of CFAs, and defines the CFA analyses that we use in Sec. 3 to describe our approach.

2.1 Control-Flow Automaton (CFA)

We borrow the definition of CFAs from Beyer et al. [5]. Since the original definition does not handle function calls, we extend CFAs to allow us to express and reason about libraries and clients. In particular, we make five small changes: (M_1) add a final location; (M_2) add a vector of inputs; (M_3) add a vector of outputs; (M_4) allow calls to other CFAs; and (M_5) allow more data types than just rational numbers. We use (M_1)–(M_3) to enable (M_4). We use (M_4) to formalize the notions of client and library: clients are CFAs that call other CFAs; libraries are CFAs that are called by other CFAs.

Syntax. A CFA $(L, l_i, l_f, \vec{x}, \vec{r}, G)$ is a finite set of program locations L, an initial location l_i, a final location l_f, a vector of input variables $\vec{x}$, a vector of output variables $\vec{r}$, and a finite set $G \subseteq L \times O \times L$ of control-flow edges. The set O of program operations consists of assignment and assumption operations. Assignments are denoted by $x \leftarrow t$, where x is a variable in $\vec{x}$ or $\vec{r}$ and t is a *term* of the same type. Assumptions are denoted by $[b]$, where b is a Boolean term. The set of terms is defined inductively. Every constant is a term, every variable in $\vec{x}$ or $\vec{r}$ is a term, and if f is a CFA and $\vec{t}$ is a vector of terms matching the input type of f, then $f(\vec{t})$ is a term. In other words, a call to the CFA f with the arguments $\vec{t}$ is a term, and this term is represented by $f(\vec{t})$.

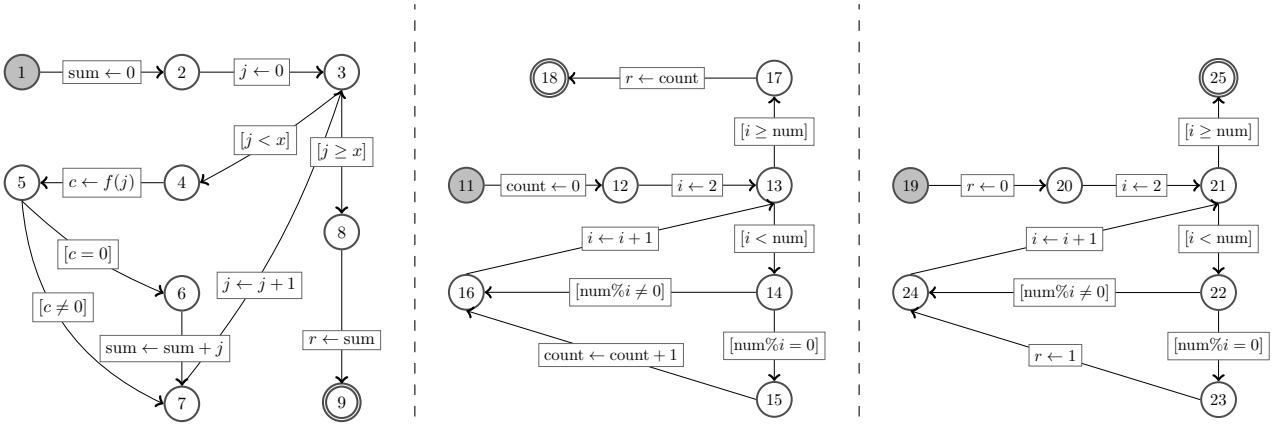
For example, the CFAs of the C functions in Fig. 1 are depicted in Fig. 2, where locations are numbered nodes, initial locations are shaded, final locations are double circled, and operations appear as labels on edges. In particular, the CFA of the client, `sum_primes`, has $L = [1, 10]$, $l_0 = 1$, $l_f = 10$, $\vec{x} = \langle x \rangle$, $\vec{r} = \langle r \rangle$, and uses f as a placeholder for either `composite_0` or `composite_1`.

Semantics. Let $f = (L, l_i, l_f, \vec{x}, \vec{r}, G)$ be a CFA. A *concrete state* of f is a pair (l, σ), where $l \in L$ is a location and σ is a variable assignment. When the location of a concrete state is obvious or irrelevant we abuse notation and omit it. Every edge $g \in G$ defines a transition relation between concrete states $\xrightarrow{g} \subseteq C \times \{g\} \times C$, where C is the set of all concrete states of f. For any two concrete states, a and b, we write $a \xrightarrow{g} b$ if $(a, g, b) \in \xrightarrow{g}$, and we use $a \rightarrow b$ to mean that there exists some $g \in G$ such that $a \xrightarrow{g} b$ holds. For example, for the CFA of `composite_1` in Fig. 2c,

$$a_{20} = (20, \{\text{num} \mapsto 0, r \mapsto 0, i \mapsto 0\})$$
$$b_{21} = (21, \{\text{num} \mapsto 0, r \mapsto 0, i \mapsto 2\})$$

are concrete states, $g = (20, i \leftarrow 2, 21)$ is a control-flow edge, and we can write $a_{20} \rightarrow b_{21}$, since $a_{20} \xrightarrow{g} b_{21}$ holds.

Let $c_1, e_1, ..., e_{n-1}, c_n$ be a sequence of alternating concrete states and edges such that $\bigwedge_{i=1}^{n-1} c_i \xrightarrow{e_i} c_{i+1}$ holds. We call $c_1, e_1, ..., e_{n-1}, c_n$



(a) `sum_primes` CFA. (b) `composite_0` CFA. (c) `composite_1` CFA.

Figure 2: CFAs corresponding to C functions in Fig. 1, where locations are nodes, initial locations are shaded, final locations are double circled, operations are labels on edges, and `f` stands for either a call to the CFA in 2b or to the CFA in 2c.

a *trace* if the location of c_1 is l_0 and the location of c_n is l_f. We call $c_1, e_1, ..., e_{n-1}, c_n$ a *run* otherwise. For a trace $c_1, e_1, ..., e_{n-1}, c_n$ we call the pair (c_1, c_n) an *i/o pair* and denote it $c_1 \Rightarrow c_n$. In other words, if $c_1 \Rightarrow c_n$ is an i/o pair of f, then executing f starting at c_1 will result in c_2. For example, for the CFA of `composite_1` in Fig. 2c and concrete states

$$c_{19} = (19, \{\text{num} \mapsto 0, r \mapsto 0, i \mapsto 0\})$$
$$c_{25} = (25, \{\text{num} \mapsto 0, r \mapsto 0, i \mapsto 2\})$$

$c_{19} \Rightarrow c_{25}$ is an i/o pair because 19 is the initial location, 25 is the final location, and there is a sequence of concrete states that connects c_{19} and c_{25} (following the sequence of locations 19, 20, 21, 25).

Given a concrete state σ, the term $f(\vec{t})$ denotes $\sigma''(\vec{r})$, where $\sigma' \Rightarrow \sigma''$ is an i/o pair of f and $\sigma'(\vec{x}) = \sigma(\vec{t})$ is true. In other words, $f(\vec{t})$ represents the return value of a call to f with the arguments $\vec{t}$, using call by value semantics. For example, at the concrete state $(4, \{j \mapsto 0, ...\})$ in the CFA of `sum_primes`, the meaning of the term $f(j)$ is 0, regardless of the version of `composite` that you plug in for f (both `composite_0` and `composite_1` will return 0 when given the input 0).

2.2 Formal Problem Definition

The CSEC problem is that of determining whether a change to a library affects its calling client. In this paper, we focus on functional effects: the input/output behaviour of the client.

Formally, let f, f', g, and g' be CFAs such that (1) f calls g, and (2) f' is the same as f but with calls to g replaced by calls to g'. We call f the *client*, and we call g and g' two versions of the *library*. We say that g and g' are *functionally client-specific equivalent (CSE)* for f iff $\forall \sigma, \vec{t}\ f(\vec{t}) = f'(\vec{t})$. In other words, we say that the libraries are CSE for the client, if the input/output behaviour of the client is unaffected by the version of the library that it uses.

The general functional equivalence checking problem is the same as CSEC but without restrictions (1) and (2). In other words, the general functional equivalence checking problem is to check whether two functions will always return the same output when given the same input. This means that we can use equivalence checkers to solve the CSEC problem. However, we argue that using the CSEC problem restrictions improves performance and makes many previously infeasible cases solvable. For example, our approach fully automatically proves that `composite_0` and `composite_1` are CSE for `sum_primes`, while ModDiff, SymDiff, RVT, and Rêve all fail to prove the corresponding general equivalence checking problem within a day.

There are two crucial insights behind our approach. First, we often only need a portion of the client to prove that the client is unaffected by the library change. We call such a portion of the client an *impact boundary*. Second, certain candidate impact boundaries are (relatively) easy to check. Our approach combines these insights by searching the CFA of the client for (relatively) easy to check candidate impact boundaries. The search stops when it finds a true impact boundary or a counterexample to equivalence. This search requires a guarantee that the client remains unchanged.

2.3 CFA Properties

Before describing our approach, we provide a few CFA definitions culminating in the definition of a *hammock*. Hammocks are parts of a CFA that are "extractable." That is, there is a semantics preserving transformation that replaces hammocks with calls to new, standalone CFAs. We use hammocks to define the structure of impact boundaries in Sec. 3.

The first set of definitions—*path*, *simple path*, *domination*, *post-domination*, *backedge*, and *reducible*—are standard graph theory terms used in program analysis [1]. Let f be a CFA $(L, l_i, l_f, \vec{x}, \vec{r}, G)$. A *path* in the CFA is a sequence of locations in L that are connected by edges in G. A path is a *simple path* if no location appears more than once in the path. We say that a location $i \in L$ *dominates* a location $j \in L$ if every path from l_0 to j passes through i. We say that a location $j \in L$ *post-dominates* a location $i \in L$ if every path from i to l_f passes through j. An edge $j \rightarrow i$ is a *backedge* if i dominates j. A graph G is *reducible* if G becomes acyclic after removing all backedges.

For example, in Fig. 2a, the label 3 dominates the label 7, the label 8 post-dominates the label 3, the edge between 7 and 3 is a

backedge, and the entire graph is reducible, since removing the edge between 7 and 3 gives an acyclic graph.

We use the second set of definitions—*strongly connected component*, *entry point*, and *exit point*—in our impact boundary search algorithm of Sec. 4. Let $f = (L, l_i, l_f, \vec{x}, \vec{r}, G)$ be a CFA. A *strongly connected component* (SCC) S of f is the maximal sub-graph of G with the property that there is a path from every location in S to every other location in S. We call a location i an *entry point* of S if $i \in S$ and there exists a location $n \notin S$ such that $n \rightarrow i$ holds. We call a location j an *exit point* of S if $j \in S$ and there exists a location $n \notin S$ such that $j \rightarrow n$ holds.

For example, in Fig. 2a, the sub-graph containing locations 3, 4, 5, 6, and 7, and all edges between these locations, is an SCC. For this SCC, the locations 3 and 8 are the entry and exit point,respectively.

The third set of definitions—*sub-CFA* and *induced sub-CFA*—depend on the first set, and will give us the main structure of *impact boundaries*. Let $f = (L, l_i, l_f, \vec{x}, \vec{r}, G)$ be a CFA. A *sub-CFA* of f is a CFA $h = (L_h, i, j, \vec{v}, \vec{v}, G_h)$ such that $L_h \subseteq L$, $j \in L_h$, $i \in L_h$, $\vec{v} \subseteq \vec{x} \cup \vec{r}$, and $G_h \subseteq G$. If $\vec{v} = \vec{x} \cup \vec{r}$ and G_h is exactly the set of edges of f that connect pairs of locations in L_h, then we say that h is the *sub-CFA* of f *induced* by L_h, i, and j. When G_h is an SCC of G, the entry point of G is i, and the exit point of G is j, then we call h an SCC of f.

For example, for the CFA in Fig. 2a, the sub-CFA induced by the subset of locations $\{1, 3, 7\}$, initial location 3, and final location 1 does not contain the locations 2, 4, 5, 6, 8 or 9, or any edges touching them. In fact, it only contains the edge from 7 to 3. This example demonstrates the problem with induced sub-CFAs: they are too unrestricted for our needs. This can make their behaviour too hard to reason about and make the guarantees they can give too weak.

The final definition, *hammock*, comes from Komondoor and Horwitz [22], but is adapted to our context. Hammocks are related to the *foldable sub-graphs* of Lakhotia and Deprez [26]. Intuitively, both are portions of a CFA that can be extracted into a standalone CFA. A sub-CFA $h = (L_h, i, j, \vec{v}, \vec{v}, G_h)$ induced by L_h, i, and j is a *hammock* iff the following hold: (1) for every $l \in L_h$, i dominates l in f; (2) for every $l \in L_h$, j post-dominates l in f; (3) every location on every simple path from i to j is in L_h; and (4) for every $l \in L_h$, every simple path in f between j and l contains i.

For example, for the CFA in Fig. 2a, we can define a sub-CFA induced by the subset of locations $\{3, 4, 5, 6, 7\}$, initial location 3, and final location 3. This induced sub-CFA is a hammock since 3 dominates and post-dominates exactly the locations 4, 5, 6, and 7, there is no "missing" location that can be reached from the initial location and leads to the final location, and there is no path from the final location to the sub-CFA that evades the initial location.

Intuitively, a hammock is easier to reason about because it acts like a a standalone CFA. We use this fact in the soundness proof in Sec. 3. Specifically, we use the fact that if h is a hammock of f, then all traces of f are of the form

$$(l_0, \sigma_0), ..., ((i, \sigma_i^k), ...(j, \sigma_j^k), ...)^*, ...(l_f, \sigma_f),$$

where * is the standard Kleene star, (i, σ_i^k) and (j, σ_j^k) are the k^{th} concrete states at location i and j, respectively, and locations i and j do not appear in any ellipses. In other words, every trace of f has zero or more runs with locations in h, and every such run starts at i and always leaves h exactly at the location j. Since every trace of f has these properties, we can think of the runs from i to j as calls to an external CFA. We use this trace pattern to reduce the analysis of a trace of f to one of the candidate impact boundary runs inside.

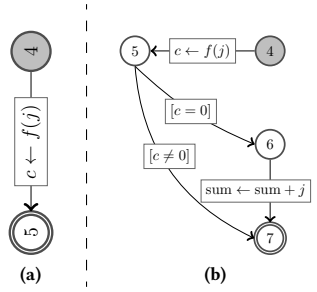


Figure 3: The two smallest candidate impact boundaries of `sum_primes` from Fig. 2a. Both are induced sub-CFAs.

3 2CLEVER AT A HIGH LEVEL

In this section, we formally define *impact boundaries*, provide a simple impact boundary search strategy, and prove the soundness and relative completeness of our approach. The completeness is relative because our search strategy depends on an equivalence checking oracle. Intuitively, our approach decomposes the CFA of the client into smaller equivalence checking queries. This query size reduction can often make intractable problems tractable.

For simplicity of presentation, we assume that every client CFA contains only one call to the library, and that every client CFA is reducible. We relax the first assumption in Sec 4, and note that the second assumption is innocuous since irreducible control-flow is rare in practice [33]. For the remainder of this section, let g be the old version of the library CFA, g' be the new version of the library CFA, and f be the client CFA with a single library call on an edge e.

3.1 Impact Boundaries

A *candidate impact boundary* $h = (L_h, i, j, \vec{v}, \vec{v}, G_h)$ is a hammock of $f = (L, l_0, l_f, \vec{x}, \vec{r}, G)$ such that h contains e. Intuitively, a candidate impact boundary is a portion of the client with the special property that if the candidate impact boundary is unaffected by the library change, then the client is unaffected by the library change.

An *impact boundary* is a candidate impact boundary h such that g and g' are CSE for h. For example, Fig. 3 depicts the two smallest candidate impact boundaries of `sum_primes` from Fig. 1a. The CFA in Fig. 3b is a true impact boundary, while the CFA in Fig. 3a is not.

3.2 Impact Boundary Search

We call our boundary search algorithm 2clever. It takes in a client and two versions of one of the client's libraries. It returns "CSE" if it finds an impact boundary and "Not CSE" otherwise. In this section, we describe a simple realization of 2clever, called 2clever-naive, and reason about its correctness and completeness. The more sophisticated realization is described in Sec. 4.

2clever-naive enumerates every sub-CFA of the client, filters out those that are not candidate impact boundaries, and then checks each candidate for CSE using an existing equivalence checker. For

example, when given the client and libraries from Fig. 2, 2clever-naive will enumerate the two candidate impact boundaries in Fig 3 (among others), check each for CSE, and return "CSE", since the CFA in Fig. 3b is a true impact boundary. In the next section, we prove the soundness of this strategy. For now, we assume correctness and note that it is cheaper to check if `composite_0` and `composite_1` are CSE for the CFA in Fig. 3b using an existing equivalence checker than to check the original problem. In fact, the CFA in Fig. 3b corresponds to removing an entire unbounded while loop from the original CFA in Fig. 2a.

3.3 Analysis

In this section, we prove two theorems about 2clever: if 2clever claims g and g' are CSE for f, then they truly are (soundness) and if g and g' are not CSE for f and our equivalence checking oracle terminates on every query, then 2clever will terminate (relative completeness).

Theorem 1 (Soundness). *If 2clever finds an impact boundary, then g and g' are CSE for f.*

Proof. Let $f = (L, l_0, l_f, \vec{x}, \vec{r}, G)$ be the client, let f' be f but with the call to g replaced by a call to g'. Similarly, let $h = (L_h, i, j, \vec{v}, \vec{v}, G_h)$ be the impact boundary, and let h' be h but with the call to g replaced by a call to g'. By the definition of candidate impact boundaries, all traces of f and f' are of the form

$$(l_0, \sigma_0), ..., ((i, \sigma_i^k), ...(j, \sigma_j^k), ...)^*, ...(l_f, \sigma_f) \quad (T_1)$$

$$(l_0, \theta_0), ..., ((i, \theta_i^k), ...(j, \theta_j^k), ...)^*, ...(l_f, \theta_f). \quad (T_2)$$

We want to prove that if $\sigma_0 = \theta_0$ then $\sigma_f = \theta_f$.

Since the client is unchanged apart from the call to the library, we know that if $\sigma_0 = \theta_0$ then $\sigma_i^1 = \theta_i^1$. We also know that, for any m and n, if $\sigma_j^m = \theta_j^n$ then every concrete state in (T_1) after (j, σ_j^m) is guaranteed to be equal to the corresponding concrete state in (T_2) after (j, θ_j^n) until both traces reach a concrete state with location i. In other words, all the "code" outside of the impact boundary remains untouched, and two copies of syntactically identical code that start at the same concrete state will produce identical traces. Given these two facts, we need to prove that the runs

$$(i, \sigma_i^k), ...(j, \sigma_j^k) \quad (T_3)$$

$$(i, \theta_i^k), ...(j, \theta_j^k) \quad (T_4)$$

inside of (T_1) and (T_2), respectively, are equal. By the semantics of function application, every run (T_3) is equivalent to the term $h(\vec{v})$. Similarly, every run (T_4) is equivalent to the term $h'(\vec{v})$. Since g and g' are CSE for h, $h(\vec{v}) = h'(\vec{v})$; therefore, (T_3) and (T_4) are equal. □

Theorem 2 (Relative Completeness). *If g and g' are CSE for f, then there is an impact boundary. Furthermore, if every call to an equivalence checker oracle terminates, 2clever finds an impact boundary.*

Proof. If g and g' are CSE for f, then f is an impact boundary. Since the set of locations of a CFA is finite, the number of candidate impact boundaries 2clever checks is finite. □

4 REALIZING 2CLEVER

In this section, we describe the actual impact boundary search algorithm used by 2clever, a bespoke equivalence checker that we use inside the search algorithm, and a resource (time) allocation scheme to improve 2clever's chance of termination. 2clever also extends 2clever-naive to handle clients that call the library multiple times. At a high level, 2clever confirms the CSE only if it finds an impact boundary for every call-site of the library.

4.1 Impact Boundary Search Revisited

Finding and checking every candidate impact boundary is impractical for complex clients. In practice, 2clever focuses on finding and checking a subset of candidate impact boundaries. Some of these are still difficult to reason about, so we transform them to *cycle-broken candidates*. While easier to check, confirming a *cycle-broken candidate* implies confirming the impact boundary that it replaced. Given this, and since 2clever keeps the client f in the subset of candidate impact boundaries to check, the proofs of soundness (Thm. 1) and completeness (Thm. 2) still hold.

Given a candidate impact boundary, h, whose initial and final locations are the same location, l, we produce a *cycle-broken* candidate h_{cb} by separating l in h_{cb}. For example, the CFA shown in Fig. 4b is a cycle-broken candidate of the client f (see Fig. 2a) whose initial and final locations both belong to location 3 in f. Locations 3, 4, 5, 6 and 7 form a strongly connected component (shown in Fig. 4a), which is hard to check for CSE. On the other hand, the cycle-broken candidate h_{cb} breaks all cycles involving location 3 (the final location does not have outgoing edges), hence makes checking CSE on h_{cb} easier. Even though h_{cb} is technically not a sub-CFA of the client f, we still refer to it as a candidate impact boundary for the rest of the paper because it is a sub-CFA of an equivalent client $f*$ where the common location l is extended to $l \to l_i$ and $l_f \to l$.

4.1.1 Candidate Search. The extended search algorithm, `Search*` (Alg. 1), identifies a sequence of cycle-broken candidates, `BoundSeq`, for each library call-site e. `BoundSeq` includes the call-site e itself and the client f as its first and last element, respectively (line 15 and 6). To identify cycle-broken candidates for a call-site e, the algorithm first looks for the strongly connected component (SCC) S that contains e. `Search*` uses an auxiliary procedure, `Normalize`, to transform S into an SCC $S*$ with a unique entry and exit point, i. The algorithm then constructs a cycle-broken candidate h_{cb} from $S*$, and adds h_{cb} to `BoundSeq`. Finally, the algorithm updates the search context $h \leftarrow h_{cb}$ (line 13) and recursively looks for a new cycle-broken candidate in h, until h has no SCC containing e (line 8). The discovered h_{cb} is always pushed to the head of the sequence.

For example, for the client f in Fig. 2a, `Search*` identifies the SCC S in Fig. 4a containing the call-site $4 \xrightarrow{g} 5$. Location 3 is the unique entry and exit of S, and it is replaced with l_i and l_f in *cycle-broken* candidate h_{cb}, where l_i is a copy of 3 with only outgoing edge $(l_i \to 4)$, and l_f is another copy of 3 with only internal incoming edges $(7 \to l_f)$. The resulting cycle-broken candidate h_{cb} is shown in Fig. 4b. 2clever then attempts to find an SCC inside h_{cb} but fails. Finally, the call-site and the client are added to `BoundSeq`.

Algorithm 1 Search*

Require: Client CFA f and library CFAs g and g'
Ensure: BoundaryMap contains a sequence of candidate impact boundaries for each call-site

```
1:  procedure Search*(f ,g)
2:      E ← call-sites of g in f
3:      BoundaryMap ← Dict()
4:      for each e ∈ E do                          ▷ For each call-site
5:          BoundSeq ← stack                       ▷ first in last out sequence
6:          BoundSeq.push (f)
7:          h ← f                                  ▷ search on hammock h
8:          while h has SCC containing e do
9:              S ← SCC containing e in h
10:             S*, i ← Normalize(S, h)
11:             h_cb ← hammock induced from S* and i
12:             BoundSeq.push (h_cb)
13:             h ← h_cb                           ▷ update cycle-broken candidate
14:         end while
15:         BoundSeq.push (e)
16:         BoundaryMap [e] ← BoundSeq
17:     end for
18:     return BoundaryMap
19: end procedure
```

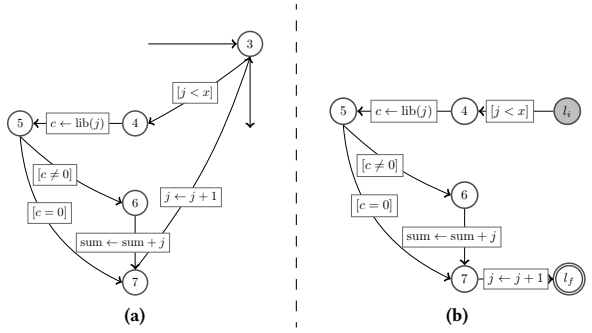


Figure 4: 4b shows the cycle-broken candidate constructed from the strongly connected component in 4a.

Theorem 3 (Correctness). *Every sub-CFA h in* BoundSeq *returned by* Search* *is a candidate impact boundary.*

Proof. BoundSeq contains the client f, the call-site e, and cycle-broken candidates h_{cb}. Both f and e are trivially candidate impact boundaries. Every $h_{cb} \in$ BoundSeq is constructed from an SCC $S*$ with a unique entry and exit i. Therefore, i dominates and post-dominates every node in $S*$, including the source and target of e. Moreover, every path from i to e must go through i. Therefore, $S*$ is a candidate impact boundary, and h_{cb} is a *cycle-broken* candidate constructed from $S*$. □

Theorem 4 (Monotonicity). *For every* BoundSeq $= [h_1 ... h_N]$ *returned by* Search*, *for all i in range $0 \leq i < N$, h_i is a candidate impact boundary for h_{i+1}.*

Proof. For $0 \leq i < N$, h_i is the cycle-broken candidate constructed from the SCC $S*$, where $S*$ is a sub-CFA of h_{i+1}. Search* ensures that $S*$ has a unique entry and exit location in h_{i+1}. Therefore, h_i satisfies the conditions of *cycle-broken* candidate for h_{i+1}. The last item in BoundSeq h_N is the client f. By correctness of Search* (Thm. 3), h_{N-1} is a candidate impact boundary for f. □

Algorithm 2 Check*

Require: BoundaryMap is returned by Search*
Ensure: $ret = \top$ iff g and g' is CSE for f

```
1:  procedure Check*(BoundaryMap, f ,g, g′)
2:      E ← call-sites of g in f
3:      Contained ← ∅
4:      for each e ∈ E do                          ▷ For each call-site
5:          BoundSeq ← BoundaryMap [e]
6:          while BoundSeq ≠ ∅ ∧ e ∉ Contained do
7:              h ← BoundSeq.pop()
8:              if EQ (h, g, g′) then
9:                  for each call-site e′_g in h do
10:                     Contained.add(e′_g)
11:                 end for
12:             end if
13:         end while
14:     end for
15:     ret ← Contained = E
16:     return ret
17: end procedure
```

Monotonicity suggests a good checking order of candidates: h_i is smaller and likely easier to check than h_{i+1}, and confirming h_i is sufficient for proving CSE.

4.1.2 Checking Candidates. After finding candidate impact boundaries with Search*, 2clever uses Check* to check each. Specifically, for each call-site e in f, Check* fetches the corresponding sequence of candidate impact boundaries, BoundSeq, from BoundaryMap (line 5), and iterates through the sequence to find a true impact boundary for e (lines 6-13). During each iteration, Check* checks whether g and g' are CSE for h by calling an equivalence checker, EQ (line 8). If EQ determines equivalence, then h is an impact boundary for every call-site e_g (including e) in h. Therefore, the algorithm adds e_g into the set Contained (line 10), which represents the set of call-sites with confirmed impact boundaries. Check* stops iterating over BoundSeq when it either reaches the end, or the target call-site e is added to Contained (line 6). Finally, the algorithm determines whether every call-site has an impact boundary by comparing Contained and E (line 15), and returns the result.

Theorem 5 (Partial Correctness of Check*). *If* Check* *terminates, then g and g' are CSE for f iff* Contained $= E$ *at line 15.*

Proof Sketch. Forward: if g and g' is CSE for f, then f is an impact boundary for all call-sites. Since f is in the BoundSeq for every call-site, then every call-site is eventually added to Contained.

Backward: e is added to Contained only if it has an impact boundary in BoundSeq. When Contained $= E$, every call-site has an impact boundary. By Thm. 1, for every call-site e, the state difference caused by visiting edge e is contained before reaching the final location l_f. □

Algorithm 3 `2clever-EQ`

Require: h is a candidate impact boundary of g.
Require: every SCC in h has single entry and single exit.
Ensure: Returns proof or counterexample.

```
1:  procedure 2clever-EQ(h, g, g')
2:      E ← call-sites of g in h
3:      callee ← Merge (E)
4:      pre-caller ← before (h, callee)
5:      post-caller ← after (h, callee)
6:      Pre ← pre_condition(summary(pre-caller))
7:      Post ← post_condition(summary(post-caller))
8:      callee× ← product(callee, callee')
9:      return verify (Pre, callee×, Post)
10: end procedure
```

4.2 Bespoke Equivalence Checking Algorithm

`Check*` calls procedure `EQ` to confirm or reject candidate impact boundaries. While any sound equivalence checker can be used for `EQ`, we developed an in-house algorithm, `2clever-EQ`, that exploits the structure of our queries to achieve the best performance.

`2clever-EQ` is given in Alg. 3. It takes the caller h and two versions of the callee, g and g', and requires that every strongly-connected component (SCC) in h has a single entry and a single exit. `2clever-EQ` first identifies all call-sites in h and then merges them into a single call-site `callee` (line 3). Specifically, the procedure `Merge` finds the smallest hammock, w, that contains all the call-sites and then returns the largest SCC containing w in h (or returns w if no SCC exists). Intuitively, h is an impact boundary for all call-sites if and only if it is an impact boundary for w, since w contains all call-sites. The SCC returned by `Merge` is the next smallest possible impact boundary and it is called at most once in h. We know the former since, by monotonicity (Thm. 4), at the time of checking h, `Check*` * has already checked the *cycle-broken* candidate constructed from `callee` (the previous candidate impact boundary in `BoundSeq`), and failed. We use the latter to partition h by `callee`, as discussed next.

`2clever-EQ` partitions h into `pre-caller` and `post-caller`: the portions of h before and after `callee`, respectively (lines 4 and 5). `2clever-EQ` then summarizes `pre-caller` and `post-caller` (lines 6 and 7) and converts the summaries into a pre-condition `Pre` and a post-condition `Post` as described in Sec. 4.2.1. `2clever-EQ` then creates a `product` function `callee×` via self-composition [3] (line 8). When the summaries of `pre-caller` and `post-caller` are defined for all possible inputs, candidate h is an impact boundary if and only if {`Pre`}`callee` × {`Post`} is a valid Hoare triple [19]. We check this Hoare triple using conditional modeling checking [6] by combining three existing program verifiers. We describe the detailed workflow in Sec. 5.

4.2.1 Summary, Pre- and Post-condition Computation. A *function summary of* f is a first-order formula φ over $\vec{\alpha}$ and $\vec{\beta}$ if

$$\forall M \models \varphi,\ (M[\vec{\alpha}] = \vec{x}) \implies (M[\beta] = f(\vec{x})),$$

where $\vec{x}$ and $f(\vec{x})$ are input and output of f, and $M[\vec{\alpha}]$ is the interpretation of $\vec{\alpha}$ in the model M. A function summary is *complete* if it is defined for all possible inputs. Similar to Clever [27], `2clever-EQ` computes summaries by symbolically executing [21] `pre-caller` and `post-caller` while recording programs' path conditions and effects. During symbolic execution, calls to `callee` are uninterrupted. If `pre-caller` and `post-caller` have finite execution paths, then the summary by symbolic execution is complete. Otherwise, other procedure summarization techniques (e.g. abstraction refinement) may be used.

Given φ^{pre}—the complete function summary of `pre-caller`—the pre-condition of `callee×` is

$$\texttt{assume}(\exists\vec{\alpha} \cdot \varphi^{pre}(\vec{\alpha}, \vec{x})),$$

where $\vec{\alpha}$ is the input of `pre-caller`, and $\vec{x}$ is the the output of `pre-caller` and the input of `callee×`. Intuitively, the pre-condition captures the possible invocations of `callee×`. Given φ^{post}—the complete function summary of `post-caller`—the post-condition of `callee×` is

$$\texttt{assert}(\forall\vec{\beta} \cdot \varphi^{post}(\vec{r}, \vec{\beta}) \leftrightarrow \varphi^{post}(\vec{r}', \vec{\beta})),$$

where $\vec{r}$ and $\vec{r}'$ are the outputs from `callee` and `callee'`, respectively, and $\vec{\beta}$ is set to be the output of `post-caller` on input $\vec{o}$ and $\vec{o}'$. Intuitively, a post-condition asserts that `post-caller` mitigates the difference, if any, in the outputs of `callee` and `callee'`.

For example, consider the candidate of impact boundary h in Fig. 3b, and the libraries g, g' in Fig. 2b and Fig. 2c, respectively. We know that h is an impact boundary because the Hoare triple, {`Pre`}`product`(g, g'){`Post`}, is valid, where the pre-condition `Pre` is `assume` $(j < x \wedge num = j)$, and the post-condition `Post` after simplification (removing $\top$ and $\bot$ from conjunctions and disjunctions, respectively) is

$$\texttt{assert}((lib(j) = lib'(j) = 0) \vee ((lib(j) \neq 0 \wedge lib'(j) \neq 0))).$$

4.3 Time-Bounded Equivalence Checking

The procedure `EQ` may not terminate due to the undecidability of equivalence checking. Since `Check*` calls `EQ` multiple times over `BoundSeq`, we need a strategy for resource (time) allocation to maximize the chance of termination. We achieve this by adding a timeout argument t to every `EQ` call, and managing t heuristically. We use the following time allocation scheme: for a candidate sequence `BoundSeq`, we start by calling `EQ` with some initial t, e.g., 30 seconds. If a timeout occurs, we double the value of t for the next call. If `EQ` produces an answer within half of t, we reduce t by half for the next call. When `EQ` is called on the last candidate f, we set t to the maximum possible value.

Intuitively, the time allocation scheme uses the result from the previous `EQ` call to predict the difficulty of the next call. We use this scheme because candidates in `BoundSeq` have monotonically increasing contexts. Therefore, the difficulty of the previous call usually correlates with the difficulty of the next one.

5 IMPLEMENTATION

We implemented a prototype of *2clever* using 3000 lines of Python code [16]. The prototype (see Fig. 5) consists of three main components, denoted ①-③: the front-end `C_to_CFA` and the implementation of algorithm, `Search*` (Alg. 1) and `Check*` (Alg. 2), respectively.

`C_to_CFA` translates C99 [20] source programs (shown as Ⓐ) to `CFA` f, g, g' (shown as Ⓑ). It uses pycparser[13] for source parsing and Abstract Syntax Tree (AST) transformation. We chose pycparser

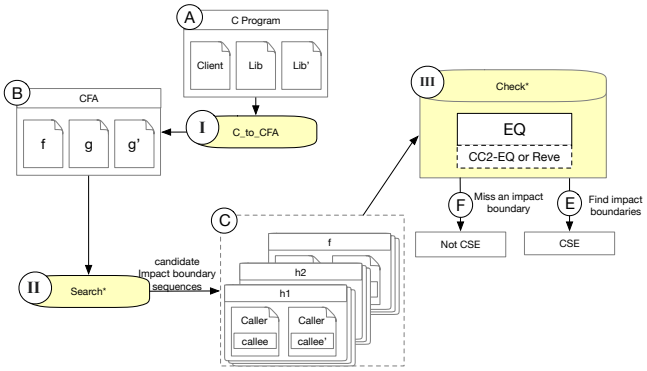


Figure 5: *2clever* architecture and workflow. *I*, *II* and *III* are the components of the architecture.

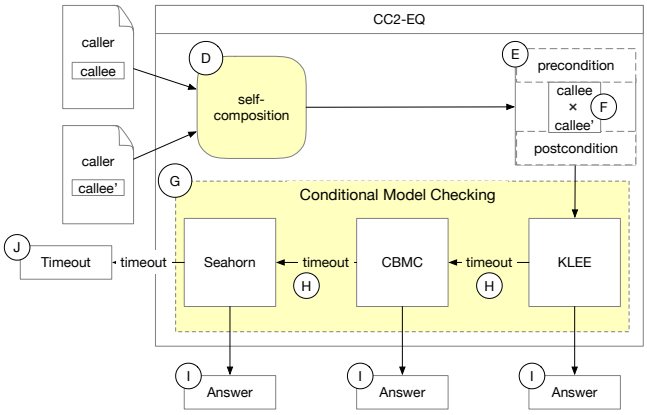


Figure 6: Internal workflow of `2clever-EQ`.

because, like our implementations of `Search*` and `Check*`, it is written in Python.

The `Search*` implementation (II in Fig. 5) identifies a sequence of candidate impact boundaries for every call-site. Analyzing the sequences allows `Search*` to identify dependent and redundant equivalence checking tasks across different call-sites when their `BoundSeq` overlap, enabling optimizations through task planning and redundancy pruning. In addition, independent equivalence checking tasks on different sequences are parallelizable.

`Check*` (III in Fig. 5) uses `EQ` for equivalence checking of individual candidate. In addition to Rêve [15], we also implement `2clever-EQ`– the in-house checker for `EQ`, illustrated in Fig. 6. `2clever-EQ` uses *conditional model checking* [6] to combine three distinct verification approaches: bounded model checking using CBMC [11], symbolic execution using KLEE [7], and IC3/PDR using Seahorn [23]. Effectively, `2clever-EQ` combines the strengths of the supported verifiers by allowing them to communicate their progress on a verification problem through a common information exchange method.

Before checking for equivalence, `2clever-EQ` applies *self- composition* [4] on the inputs to express the equivalence checking problem as a verification problem (D in Fig. 6 and line 8 in Alg. 3) against pre- and post-conditions computed from the caller's summary (shown as E). *Self-composition* facilitates mutual invariant learning on the merged program (F) for proving relational properties, and in our case, functional equivalence.

During the equivalence verification (line 9 in Alg. 3), `2clever-EQ` calls the verifiers in turn, with a timeout (G in Fig. 6). When a verifier times out, its progress is saved as program conditions and passed to the next verifier to guide the exploration towards the unverified state space (step H). The verification process terminates if `2clever-EQ` answers the equivalence question (step I) or exhausts all of the verifiers (step J).

Limitations. Our `CFA` model assumed the input functions are non-recursive (see Sec. 2) and reducible (see Sec. 3). Additional constraints may be introduced by the equivalence checker for `EQ`. The implementation of the in-house checker `2clever-EQ` has the following constraints, due to source level self-composition and conditional model checking, respectively: (1) functions must be well-structured, and they must exit from their last statement; and (2) datatypes are limited to chars, integers, booleans and non-parametric arrays. We use `2clever-EQ` as the default checker and automatically switch to Rêve for any instances that do not satisfy these constraints.

6 EVALUATION

In this section, we present a set of CSEC benchmarks, and compare our prototype implementation of 2clever to the state-of-the-art. We aim to answer the following research questions. **RQ1**: How does 2clever's performance compare to state-of-the-art on benchmarks from related work? **RQ2**: How does 2clever scale compared to state-of-the-art as the algorithmic complexity of the input programs varies? **RQ3**: How does 2clever scale compared to state-of-the-art as the number of library calls in the input programs increases?

6.1 Subjects and Setup

We compare 2clever, Clever and Rêve on 568 benchmarks. Each benchmark consists of a pair of C programs before and after some changes to the library. At a high level, Clever generates logical summaries of the client and the libraries, composes them, and checks for CSE using an SMT solver. Clever's performance depends on two main features: *eager counterexample detection* and *lazy library summarization*. The former is a method to disprove equivalence while generating the logical summaries; the latter is a method for only summarizing those parts of the libraries used by the client. The original Clever tool checks Python programs. We use a our reimplementation of Clever for C programs.

Rêve [15] translates the equivalence checking problem into a constrained Horn clause query that can be discharged by an off-the-shelf solver. At a high level, Rêve uses a solver to find coupling predicates over the two input programs. For optimal performance, we evaluate Rêve on the benchmarks with library calls inlined.

We started with 29 publicly accessible benchmarks collected from related work [27, 34] that followed the client-library format. We removed five cases that had recursive calls and included the example from Sec. 1. Most cases in the resulting set were too simple and therefore insufficient as test subjects for performance evaluation: 75% of the cases are solved by all three tools in under five

seconds. This set also has no nested loops; all loops iterate fewer than 20 times, and each case has only one library call. We thus added 15 benchmarks that Rêve's authors sourced from *glibc* implementations. These benchmarks did not fit our client-library format, but we were able to find client functions for these benchmarks on GitHub. This yielded the set *B-Orig* of 40 cases—32 equivalent and eight non-equivalent.

To further increase *B-Orig*'s difficulty, we systematically generate 528 benchmarks, referring to the resulting set as *B-Hard*. *B-Hard* consists of combinations of *B-Orig* benchmarks involving at least one applicable *B-Orig* case from the 75th percentile of the slowest cases (i.e., those that timed out or had a solution time of > 5 seconds for at least one tool). Below, we describe the two templates, `SeqMerge` and `NestMerge`, that we use to generate the various combinations.

`SeqMerge` takes an arithmetic operator $\Theta \in \{+, -\}$ two clients, f_1 and f_2, and their corresponding libraries, g_1 and g_2, respectively. `SeqMerge` returns two merged clients: $f_{m1} = f_1 \,\Theta\, f_2[g_2 \leftarrow g_1]$ and $f_{m2} = f_1[g_1 \leftarrow g_2] \,\Theta\, f_2$, where the substitution $f_1[g_1 \leftarrow g_2]$ means replacing every call to g_1 in f_1 with g_2. This process guarantees that the resulting programs have two library calls, mitigating for the absence of multiple library calls in *B-Orig*. It also creates new examples with library calls under different client contexts.

`NestMerge` takes the same input as `SeqMerge`, but returns client functions computed by $f_{c1} = f_1[g_1 \leftarrow f_2]$ and $f_{c2} = f_2[g_2 \leftarrow f_1]$. Intuitively, `NestMerge` uses one client's context to call the other client function as its library. `NestMerge` creates additional cases with library calls under more complex client contexts, and increases the variety of control flow patterns in benchmarks.

We ran experiments on Ubuntu 18.04 with an Intel® Core™ i7 CPU processor and 8 GB of RAM. Each case was run with timeout set to 300 seconds, and memory limit set to 10 GB.

6.2 RQ1: Performance of 2clever

We evaluate the overall performance of 2clever relative to Clever and Rêve by comparing their running times on all 568 benchmarks – both the 40 *B-Orig* cases and 528 *B-Hard* cases.

Results. The cactus plot in Fig. 7 shows the number of benchmark cases correctly solved by each tool under the specified time limit. Time is measured in seconds and plotted on a logarithmic scale. Under a 300-second time limit, 2clever, Clever and Rêve respectively solved 40, 21, and 19 of the 40 *B-Orig* cases. On *B-Hard* benchmarks, 2clever solved 498 of 528 cases with 30 instances timing out, Clever solved 368, and did not solve 160, while Rêve solved 231, and was unable to solve 297. Rêve timed out on 55 cases, and did not provide correct solutions on 242. Individually, 2clever, Clever, and Rêve solved 236, 179, 100 out of 264 `SeqMerge` instances, and 262, 189, 131 out of 264 `NestMerge` instances, respectively. In particular, on equivalent benchmarks, every case correctly solved by either Rêve or Clever was also solved by 2clever, with one exception. 2clever remained competitive with Clever on non-equivalent benchmarks, solving 152 vs. Clever's 156 despite the early detection of counterexamples feature of the latter. As the average solution time for cases increased, 2clever significantly outperformed Rêve and Clever. Even though CC2's static analysis for searching candidate of impact boundaries (see Alg. 1) adds a near constant time cost, its benefit in restricting client context significantly outweighs the cost on non-trivial instances, and ultimately makes CC2 scale better. Beyond performing well on our new benchmarks, 2clever also correctly handled the four original cases that contained unbounded loops, *ultra_prime_sum*, *pos*, *pos2* and *odd* [16], which no previous tool was able to solve under the given time limit. 2clever's success on these cases supports our claim that impact boundary search is more effective than prior techniques whenever input programs have unbounded loops with a non-trivial loop condition.

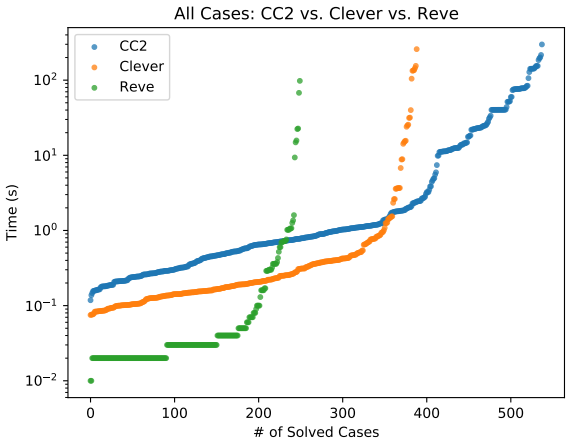


Figure 7: All benchmarks, sorted by the solution time.

Answer to RQ1. 2clever's performance on solving hard client-specific equivalence instances is superior to state-of-the-art, while its performance on easy instances remains competitive.

6.3 RQ2: Varying Program Complexity

We now study how 2clever scales with respect to the *total program complexity* of the input programs. Complexity objectively categorizes variation in control-flow patterns, while having external validity. Intuitively, we take complexity to be a rough estimate of the difficulty of equivalence checking for the benchmark cases.

To investigate, we present the results from the experiment conducted in Sec. 6.2 but this time, we sort the benchmark cases by the total benchmark complexity (as a function of N – the size of input to each program) and organize them into four complexity classes. Whenever an input program pair has two different complexities, the benchmark is assigned the higher one. The first class, denoted Constant, contains 359 $O(1)$ cases. The second class, denoted Linear, contains 100 $O(N)$ cases, two $O(NlogN)$ cases and 38 $O(logN)$ cases. The third class, denoted Quadratic, contains 50 $O(N^2)$ cases. The fourth class, denoted Higher-Order, contains 19 cases, and includes four $O(N^3)$ cases, as well as cases with non-terminating program paths. We compare the percentage of cases solved by each tool across the complexity classes.

Results. Fig. 8 displays the number and the percentage of instances solved by 2clever, Clever and Rêve, over each complexity class. The mean solution time for each tool, over all *solved* cases, is shown underneath each complexity class. Time is measured in seconds, and all cases where tools time out or do not provide a solution are

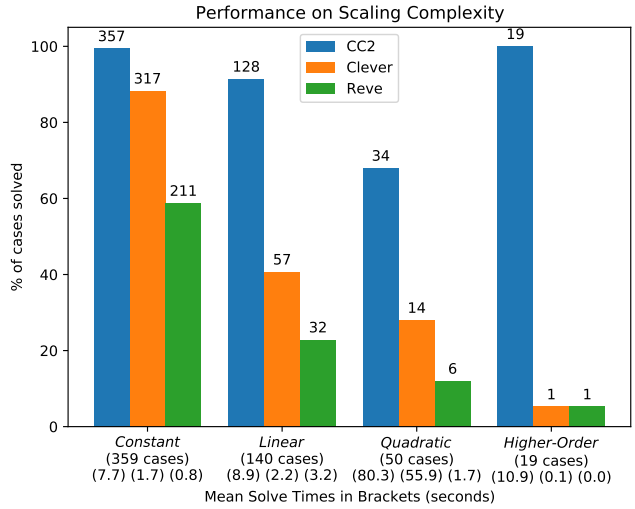


Figure 8: All tools over all benchmarks, sorted by benchmark complexity. Mean solution times are over *solved* cases.

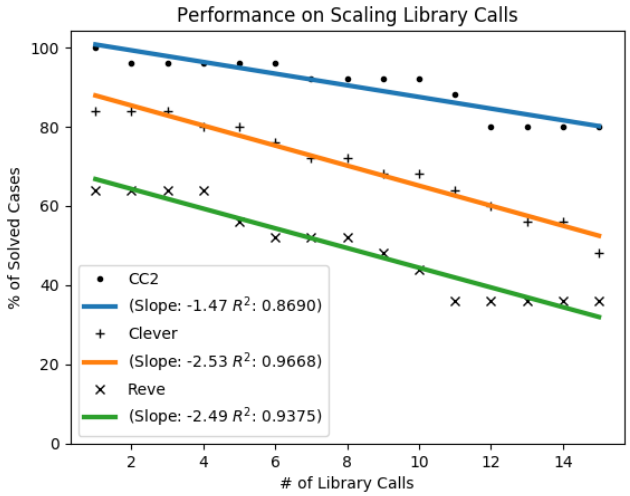


Figure 9: Scaling number of library calls over *B-MultCall*.

disregarded. We observe that 2clever outperforms Clever and Rêve on every benchmark complexity class. Although 2clever solves a smaller percentage of cases as complexity grows, as expected, its advantage over Clever and Rêve, in terms of difference in solved cases, increases. In particular, while Clever and Rêve remain competitive on lower complexity cases, 2clever significantly outperforms existing tools on higher complexity benchmarks; this provides further evidence that our impact boundary search scales more effectively to difficult cases than do Rêve and Clever. Specifically, Clever does not scale to benchmarks with higher complexities because it is path-based and terminates only when exploring a finite number of program paths is sufficient to produce an answer; this is not the case whenever unbounded loops exist anywhere in the input programs. Rêve's scalability is limited instead by its underlying reasoning engine—while verification conditions are produced quickly, the resulting monolithic queries cannot be handled by the reasoning engine. 2clever's ability to both handle unbounded client loops and decompose large queries is key to explaining its significant performance advantage on higher complexity benchmarks.

Answer to RQ2. 2clever scales more effectively than existing tools as the complexity of the input program increases. In particular, 2clever's relative performance, in terms of the percentage of solved cases, improves as complexity grows.

6.4 RQ3: Scaling Library Calls

We now study how 2clever scales as the number of library calls in the input programs increases, since real software clients can call their libraries at multiple program locations. We believe that the number of library calls correlates with the difficulty of equivalence verification because each library call site can be a potential source of difference that splits the input programs' control and data flows.

For this study, we generate another set of benchmarks with the `SeqMerge` template, focusing specifically on the number of calls. To do so, we merge *B-Orig* cases (see Sec. 6.1) with themselves, sequentially applying `SeqMerge` from one to fifteen times. This yields fifteen new cases for every applicable *B-Orig* case, with each new case containing a different number of library calls. The resulting benchmark, denoted *B-MultCall*, consists of $25 \times 15 = 375$ cases. Our experiment compares 2clever's performance to Clever and Rêve on *B-MultCall* in terms of the percentage of cases solved by each tool.

Results. Fig. 9 plots the percentage of *B-MultCall* cases solved correctly and indexed by the number of library calls in the benchmark. 2clever not only always solves the highest percentage of cases but it also extends its lead over Clever and Rêve as the number of calls increases. Specifically, 2clever's lead over Clever and Rêve grows from 16% to 36%, and from 36% to 44%, respectively. Moreover, a regression line [28] fit to each tool's performance data (see Fig. 9) suggests 2clever's rate of performance decrease is not only smaller over the observed range of library calls, but will stay smaller even outside the range. 2clever's regression line has a greater slope coefficient (-1.49) compared to Clever's (-2.53) and Rêve's (-2.49). These results show a noticeable performance advantage of 2clever compared to other tools, as the number of library calls increases.

Answer to RQ3. 2clever's performance scales better than the state of the art, in terms of the percentage of solved cases, as the number of library calls increases.

6.5 Threats to Validity

Our strategies for determining ground truth and program complexities of 528 new benchmarks are possible threats to validity.

To mitigate the absence of ground truth, two authors first manually classified equivalence of all possible combinations of client contexts and libraries from the original benchmarks, and recorded results as known facts. We then automatically established ground truth of each merged case using conservative inference rules and these known facts. For example, we say that a `SeqMerge` case (see Sec. 6.1) is equivalent if both arguments to the chosen arithmetic operator are equivalent. Similarly, we say that a `NestMerge` case (see Sec. 6.1) is equivalent if its inner function is unaffected by the library change. These inference rules applied to 229 out of 264 `SeqMerge`, and 191 out of 264 `NestMerge` cases. We determined ground truth of the remaining 35 `SeqMerge` cases and 71 `NestMerge` cases by

examining consensus, if reached, between the different tools. Disagreements and random instances were manually inspected. In cases where disagreements arose from tool's differences in definitions of equivalence (e.g., the equivalence relation over error states), or modeling semantics (e.g., whether integer overflows can occur), both answers were accepted as correct. Modelling and definitions of equivalence are important considerations, but we do not focus on them here.

We determined complexities by first analyzing the complexity of each client context and library separately, and then combining results to determine the complexity of the composed benchmark cases.

Finally, we note that our combined benchmarks could give an unfair advantage to our technique if either our original benchmark cases, or our templates for combining them (i.e., `SeqMerge` and `NestMerge`) favor it. We mitigate the former threat by generating new examples exclusively from benchmarks found in related work. We mitigate the latter by using templates, i.e., sequential and nested compositions, that reflect how real programs are constructed.

7 RELATED WORK

In this section, we describe the tools and techniques most related to 2clever. After revisiting Clever [27], we describe tools that perform general equivalence checking, dividing them into those that prioritize proving equivalence, those that prioritize disproving equivalence, and those that deal with similar input programs. We end by examining incremental verification, a related verification technique.

Client-Specific Equivalence Checking. Clever is the only other tool that specifically targets the CSEC problem. We described Clever in Sec. 6.1, and analyzed it extensively in Sec. 6. Clever is effective when there is a single library call per client path, and when a finite number of paths is sufficient to solve the CSEC problem. The main difference between 2clever and Clever is that 2clever targets more realistic programs, i.e., those with multiple library calls and non-constant time complexity.

Proving Program Equivalence. Barthe et al. [3] reduce equivalence checking to the task of verifying a product program. Many others extend this work. For a recent example, Churchill et al. [10] optimize the construction of the product program by comparing program traces. We incorporate the reduction idea into our approach (see Sec. 4.2). However, unlike Barthe et al., our formulation avoids the need for human input, and unlike Churchill et al., our main contribution is in the identification of impact boundaries.

SymDiff [24] uses mutual function summaries to check partial equivalence of two procedures by discharging verification conditions to Boogie [2]. More recent improvements [25] lessen SymDiff's user burden by automatically inferring common invariants. Unfortunately, SymDiff with this extension is unable to automatically solve our examples. We do not compare 2clever with Barthe et al., Churchill et al., or SymDiff because they do not disprove equivalence.

Disproving Program Equivalence. *Differential symbolic execution (DSE)* by Person et al. [29] uses symbolic execution to characterize the difference between two programs. DSE is similar to Clever, but it does not specifically target CSEC. *Directed incremental Symbolic Execution (DiSE)* [30] extends DSE. It uses static analysis to guide symbolic execution to areas of the program that are likely to differ. *Shadow Symbolic Execution* (SSE) [8] is similar to DiSE in that it is based on symbolic execution and it prioritizes the exploration of paths that are likely to expose a difference in the two input programs. In contrast to DiSE, SSE's heuristics are dynamic. Like Clever, none of DSE, DiSE, or SSE are able to prove equivalence of infinite path programs. For this reason, we do not compare 2clever with these tools.

Regression Verification. We described Rêve [15] in Sec. 6.1 and evaluated its performance in Sec. 6. Rêve is similar to the work by Barthe et al. [3] and Churchill et al. [10]. However, unlike the former, Rêve is fully automatic; unlike the latter, Rêve is more sensitive to the syntactic similarity of the input programs; unlike both, it is able to disprove equivalence. RVT [17] uses a fixed set of proof rules to prove the equivalence of two related programs. RVT deals with function calls bottom up, making it difficult to reason about client contexts. We do not compare against RVT because it does not disprove equivalence. ModDiff [34] extends DSE with modular symbolic execution and abstraction. ModDiff is similar to Clever, but it does not target the CSEC problem, and thus does not take advantage of the top-down exploration. We also do not compare against ModDiff because it is unable to handle programs with multiple library calls.

Incremental Verification. Incremental verification tools aim to reduce the cost of verifying a system over time by reusing verification results of previous versions [9, 12, 14, 18, 32]. For example, eVolCheck [32] maintains function summaries—logical over-approximations of the input system's functions that satisfy the system's specification—that are updated if necessary for each new version of the system. The goal of incremental verification tools is related to client-specific equivalence checking, but the problem differs in that it requires specifications.

8 CONCLUSION

In this paper, we defined the notion of impact boundary, and presented an algorithm, 2clever, that solves the functional CSEC problem by searching for impact boundaries. We implemented a prototype for 2clever and compared it against the state-of-the-art on a novel set of 568 benchmarks. We found that 2clever's performance scales better than the state-of-the-art in terms of the computational complexity and number of library calls of the input programs.

In the future, we intend to extend 2clever's applicability, scalability, and interface even further. Specifically, we aim to support the analysis of recursive programs and programs containing variability. In terms of scalability, we intend to equip 2clever with parallel solving capabilities. When candidate impact boundaries do not overlap, they can be checked independently, in parallel, with a potential of improving performance on inputs with more than one library call. 2clever's naive parallelization approach is promising [16], and we intend to further our explorations in this direction. Finally, in terms of interface, we intend to study how 2clever can use impact boundaries to communicate the "reason" for equivalence to client developers.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques.* Addison Wesley.

[2] 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.* Springer Berlin Heidelberg, 364–387.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Proc. of FM'11.* Springer Berlin Heidelberg, 200–214.

[4] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2011. Secure Information Flow by Self-Composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

[5] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. 2018. *Combining Model Checking and Data-Flow Analysis.* Springer International Publishing, Cham, 493–540.

[6] Dirk Beyer, Thomas A Henzinger, M Erkan Keremoglu, and Philipp Wendler. 2012. Conditional Model Checking: A Technique to Pass Information Between Verifiers. In *Proc. of SIGSOFT FSE'12.* ACM.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. of OSDI'08.* USENIX Association, Berkeley, CA, USA, 209–224.

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

[9] 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.

[10] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proc. of PLDI'19.* ACM, New York, NY, USA, 1027–1040.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. of TACAS'04.* Springer, 168–176.

[12] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. 2005. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *Proc. of CAV'05.* Springer, 449–461.

[13] Eli Bendersky. 2019. Complete C99 Parser in Pure Python. https://github.com/eliben/pycparser.

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

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

[16] N. Feng, V. Hui, F. Mora, and M. Chechik. 2020. CC2. https://github.com/CC24LIFE/CC2.

[17] 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.

[18] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco AA Sanvido. 2003. Extreme Model Checking. In *Proc. of VSTTE'03.* Springer, 332–358.

[19] Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

[20] ISO. 1999. *ISO/IEC 9899:1999: Programming Languages — C.*

[21] James C King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[22] Raghavan Komondoor and Susan Horwitz. 2000. Semantics-Preserving Procedure Extraction. In *Proc. of POPL'00.* 155–169.

[23] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-Based Model Checking for Recursive Programs. *Formal Methods in System Design* 48, 3 (2016), 175–205.

[24] 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.

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

[26] Arun Lakhotia and Jean-Christophe Deprez. 1998. Restructuring Programs by Tucking Statements into Functions. *Information and Software Technology* 40, 11-12 (1998), 677–689.

[27] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-Specific Equivalence Checking. In *Proc. of ASE'18* (Montpellier, France). ACM, 441–451.

[28] Raymond H. Myers and Raymond H. Myers. 1990. *Classical and Modern Regression with Applications.* Vol. 2. Duxbury Press, Belmont, CA.

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

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

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

[32] 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.

[33] James Stanier and Des Watson. 2012. A Study of Irreducibility in C Programs. *Software: Practice and Experience* 42, 1 (2012), 117–130.

[34] 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.