Automated Software Transplantation

Earl T. Barr Mark Harman Yue Jia Alexandru Marginean Justyna Petke
CREST, University College London, Malet Place, London, WC1E 6BT, UK
{e.barr,m.harman,yue.jia,alexandru.marginean.13,j.petke}@ucl.ac.uk

ABSTRACT

Automated transplantation would open many exciting avenues for software development: suppose we could autotransplant code from one system into another, entirely unrelated, system. This paper introduces a theory, an algorithm, and a tool that achieve this. Leveraging lightweight annotation, program analysis identifies an organ (interesting behavior to transplant); testing validates that the organ exhibits the desired behavior during its extraction and after its implantation into a host. While we do not claim automated transplantation is now a solved problem, our results are encouraging: we report that in 12 of 15 experiments, involving 5 donors and 3 hosts (all popular real-world systems), we successfully autotransplanted new functionality and passed all regression tests. Autotransplantation is also already useful: in 26 hours computation time we successfully autotransplanted the H.264 video encoding functionality from the x264 system to the VLC media player; compare this to upgrading x264 within VLC, a task that we estimate, from VLC's version history, took human programmers an average of 20 days of elapsed, as opposed to dedicated, time.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Automated software transplantation, autotransplantation, genetic improvement

1. INTRODUCTION

Software engineers spend a great deal of time extracting, porting, and rewriting existing code to extend the functionality of existing software systems. Currently, this is tedious, laborious, and slow [15]. The research community has provided analyses and partial support for such manual code reuse: for example, clone detection [5, 32, 36, 60], code migration [38, 58], code salvaging [12], reuse [13, 14, 16], dependence analysis [3, 22, 29, 30] and feature extraction techniques [24, 33, 44]. However, the overall process remains largely unautomated, particularly the critical transplantation of code that implements useful functionality from a *donor* into a target system, which we call the *host*.

What if we could automate the process of extracting functionality from one system and transplanting it into another? This is the goal we set ourselves in this paper. That is, we are the first to develop and evaluate techniques to implement automated software transplantation from one system to another, which one of the authors proposed (hitherto unimplemented and unevaluated) in the keynote of the 2013 WCRE [28].

A programmer must first identify the entry point of code that implements a feature of interest. Given an entry point in the donor and a target implantation point in the host program, the goal of automated transplantation is to identify and extract an *organ*, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. The programmer also supplies test suites that guide the search for donor code modifications required to make it fully executable (and pass all test cases) when deployed in the host.

This is a challenging vision of transplantation, because code from one system is unlikely to even compile when it is re-located into an unrelated foreign system without extensive modification, let alone execute and pass test cases. The extraction of the code also involves identifying all semantically required code and the successful insertion of the code organ into the host requires nontrivial modifications to the organ to ensure it adds the required feature without breaking existing functionality. In this paper, we present results that demonstrate that this vision of automated software transplantation is indeed achievable, efficient and potentially useful.

Feature identification is well studied [14, 16, 58]. Extracting a component of a system, given the identification of a suitable feature is also well studied, through work on slicing and dependence analysis [22, 24, 29, 33, 44]. The challenges of automatically extracting a feature from one system, transplanting it into an entirely different system, and usefully executing it in the organ beneficiary, however, have not previously been studied in the literature.

We developed μTrans, an algorithm for automated software transplantation, based on a new kind of genetic programming, augmented by a novel form of program slicing. μTrans synergizes analysis and testing: analysis extracts an 'organ', an executable slice from a donor; testing guides all phases of autotransplantation — identifying the organ in the donor, minimizing and placing the organ into an ice-box, and fi-

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771796

nally, specializing the organ for implantation into a target host. Our slicing technique grows an organ in the donor until that organ passes a test suite, thereby incorporating the functionality that the test suite exercises. To increase the performance and functional coverage of the test suite, we introduce *in-situ* testing, a novel form of testing that constrains the input space of traditional testing to more closely approximate behaviourally relevant inputs.

We implemented our approach in μSCALPEL and evaluated it in 15 transplantation experiments, using five donor programs and three host programs. Our evaluation is systematic: each and every donor is transplanted into each and every host. As a sanity check, we also transplanted one donor into itself. All experiments are repeated 20 times, as the underlying algorithm, which is based on genetic programming, is stochastic.

All the systems involved in our experimentation are real-world systems in current use, ranging in size from 0.4k lines of code to 25k lines of code for the donor and 25–363k for the host. We report on the effectiveness of transplantation along two axes. First, we ask whether transplantation disrupts the host's existing functionality by means of regression testing and second, whether the postoperative host can execute the transplanted software organ and exhibits nontrivial, new functionality. We use an acceptance test suite to verify (on output not exit code [49]) the organ's behavior.

We close with a case study of the transplantation of an encoder for the H.264 video encoding standard from the x264 encoder into the VLC media player. The maintainers of VLC are downstream users of x264. This encoder changes every couple of years. For VLC, x264 consists of a binary library and a wrapper that translates VLC's internal protocols into the x264 API. To keep up, the VLC community must periodically replace their x264 library, including functionality in it that VLC does not use, and update their wrapper. Across 39 updates of x264 over 11 years[1], updating VLC for the new version of x264 averaged 20 days of elapsed time. In contrast, μSCALPEL, supplied with the encoder's organ entry point and an insertion point in the host, took only 26 hours. Obviously, it does not make sense to transplant an organ into a host that already has that organ. Thus, we stripped any reference to x264 from VLC by building it with `./configure --enable-x264=no`. Under this setting, the x264 library is not linked and the preprocessor eliminates VLC's x264 wrapper. This setup allowed us to produce a postoperative version of VLC that dispenses with the wrapper and slices out unused portions of the x264 library, dropping 47% of the x264's source code. This case study demonstrates that software transplantation can be automated and can transplant real software functionality that is both practical and useful.

Our experimental results provide empirical evidence that software transplantation is realistic and viable. In 12 out of 15 experiments, involving 5 donors and 3 hosts, we successfully transplanted functionality that passes all regression and acceptance tests. Moreover, μSCALPEL autotransplanted a H.264 multi-threaded encoding feature from x264 into the VLC media player in only 26 hours.

Transplantation is a longstanding development practice. An organ is existing functionality with a well-defined interface. Currently, programmers must manually identify an organ, extract it, transform it so that it executes in the host, then validate the correctness of the postoperative host. Identifying an organ involves identifying a donor, an organ's entry point in that donor, a host, and an implantation point in that host. These are hard problems μSCALPEL does not address. Stepping in after these problems have been solved, μSCALPEL automates the extraction of the organ, *i.e.* the identification of its lines from its entry point, and its implantation into the host. In addition to an organ's entry point, μSCALPEL also requires a test suite for the organ and one for the host. Given these inputs, μSCALPEL computes an *over-organ*, an over-approximate, context-insensitive slice over the donor's call graph; implements observational slicing [8] in GP to reduce the organ, transform it to execute in the host, and validate the postoperative host. Our approach automates the later stages of software transplantion, making it faster, less laborious and cheaper. Automated transplantation promises new opportunities for enhancing development practices. For example, *speculative transplantation* may allow the exploration of feature reuse across systems that go untried because of cost.

[1]We identified these 39 updates by manually inspecting VLC's version history, so this number is a lower-bound.

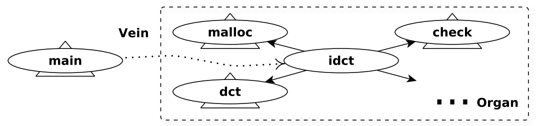


Figure 1: The call graph identifying the vein and the organ. The wavy line is the vein; `main` is the donor's entry point; `idct` is the organ's entry; the rest of the nodes are on the forward slice. A triangle behind a node suggests subgraphs reachable from that node.

The main contributions of this paper are:

1. The formalization of the software transplant algorithm (Sec. 3);
2. The first software transplant algorithm μTrans and its realization in μSCALPEL (Sec. 4); and
3. A demonstration of μSCALPEL's effectiveness, over real-world programs, at transplanting software organs — useful, nontrivial functionality — from donors to hosts (Sec. 6 and Sec. 7).

2. MOTIVATING EXAMPLE

For us, an organ is code whose functionality we wish to reuse by copying it to and adapting it for a new host. We assume that an organ starts at a function, which a user annotates with `/* OE */`. One such organ is the function `idct` in Fig. 1, which computes the inverse discrete cosine transform coefficients for an array of integers. Although there is little reason to move `idct` into cflow, we move `idct` from the donor, an audio streaming client, to cflow to illustrate how μSCALPEL works[2]. To mark `idct` as our organ, we change line 36 in file `dct.c` to '`short *idct (int *idata){ /*OE */`'.

Given a test suite, a donor with an annotated organ entry point, and a host with an annotated implantation point, μTrans solves two problems: identifying an organ within the donor and mapping that organ's variables to the host's variables at the implantation point.

We use slicing to identify the organ and one of its veins. To find a vein, we backward slice from the given organ entry point until we reach the function `main`, then prune the slice to a single path. To find the organ, we slice forwards. To compute these slices, we context-insensitively traverse the donor's

[2]`http://crest.cs.ucl.ac.uk/autotransplantation/downloads/Idct.tar.gz` contains this example; we renamed its variables here for readability.

call graph and transitively include all the functions called by any function whose definition we reach. Fusing the slices produces an over-organ that conservatively over-approximates the actual organ. To be viable, automatic transplantation cannot use the over-organ. To solve this problem, μSCALPEL builds an organ that includes only those portions of the over-organ needed to pass a user-supplied organ test suite. For example, '`fwrite (prev_samples, 2, num_samples, ofile)`' (`recv.c`:64) outputs an audio file. The organ test suite ignores this functionality, so μSCALPEL removes it.

An organ requires a specific execution environment. To make the required program state explicit in the organ's signature, we lift globals in the unmodified organ in the donor into its signature. The signature for our running example contains the following local and global variables: '`int N, double scale0, FILE *ifile, FILE *ofile, ...`'.

The insertion point in the host defines an execution environment. We choose `output.c`:342 as the implantation point because it exposes the array of line numbers for the organ. After annotation `output.c`:342 is '`static void tree_output(){ /*IP*/`'. While deciding where to place them may be difficult, `OE` and `IP` are the only two annotations μSCALPEL requires.

We reify the host environment as an interface that captures all variables, both local and global. Transplantation requires mapping the parameters of this host-side interface to the organ's parameters. We can consider the organ's signature and the graft interface as set of parameters whose elements are typed formals. Under this interpretation, transplantation fails if the parameter set of host's graft interface is not superset of the organ's signature. The problem then is to efficiently find an ordering of a subset of the parameters in the graft interface that matches the organ's signature and is correct under an organ's test suite. For this, μSCALPEL binds the variables in the organ's signature at the parameters of `graft_idct` and synthesizes a call. For our example, this call is: '`graft_idct(length,symbols,i,outfile,num,...)`'.

3. PROBLEM FORMULATION

This section defines the software transplantation problem; Sec. 4 shows μTrans, our algorithm for solving this problem.

Terminology D is a donor program from which we seek to extract a software organ O, code in D that implements some functionality of interest, to transplant. The organ O has entry points or locations, which we call L_O. H is the host program into which we seek to transplant O at L_H, the target location(s) in the host. From the start of D, there may be many paths of execution that reach one of L_O; there is at least one, or O is nonfunctional and not worth transplanting. These paths build the program environment, Γ_O, on which O's execution depends. For instance, Γ_O may contain initialised globals or populated global data structures. In keeping with the transplantation analogy, we call these paths veins. We assume that all of them correctly build Γ_O, pick one, and call it V.

Software Organ Transplantation Organ transplantation involves inserting all statements from the organ O and V into the target site in the host. For the host, H, with target site L_H, and the organ to be transplanted, O, the postoperative host is $H' = \phi(H, L_H, O)$. The function ϕ is a set of rewriting rules that adapt O to the lexical context defined by L_H.

The output of O in the donor D may differ from its output in H'. For example, assume O sorts and outputs a set of numbers from a file and that D merely opens the file before

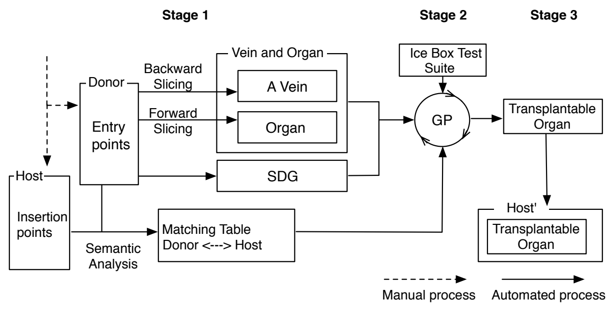


Figure 2: Overall architecture of μTrans; an SDG is a system dependency graph that the genetic programming phase uses to constrain its search space.

invoking O, while H' opens and doubles the contents of the file before invoking O. We therefore need to equate observations that can be made of executions of O in the donor D to those in H'. We denote this equality $\simeq$. It can be thought of as a mapping from an oracle for O in the donor to an oracle for O in the postoperative host.

To be successful, a transplanted organ must 1) not disrupt the existing functionality of its host and 2) must actually execute and add the desired functionality to its host.

Definition 1 (Software Organ Transplantion)**.** *Software organ transplantation* has occurred when $[\![H]\!] = [\![H']\!]_{\neg O} \wedge [\![D]\!]_O \simeq [\![H']\!]_O$, where $[\![P]\!]_O$, borrowed from denotation semantics [55], is the meaning of P restricted to the observable behaviors of O in P.

μTrans Transplantation Program equivalence is undecidable, so we approximate it with testing. To realize Def. 1, we first regression test H' against H's test suite, then use acceptance testing over tests that elicit O's behavior in H'.

Let T_R^H be the regression test suite for a host H. We say that the organ O passes regression testing if (and only if) all regression test cases yield identical observations on the original host H and the postoperative host H':

Definition 2 (Regression Testing)**.** The organ beneficiary H' *passes* regression testing iff $\forall t \in T_R^H \cdot H(t) = H'(t)$.

Let T_A^D be the acceptance test suite for D that exercises the organ O. We say that an organ beneficiary passes acceptance testing if (and only if) all of the donor's acceptance test cases that exercise O yield observations that are equivalent under $\simeq$ to those same test cases adapted as necessary, by the function τ, for the organ beneficiary:

Definition 3 (Acceptance Testing)**.** The organ beneficiary H' *passes* acceptance testing iff $\forall t \in T_A^D \cdot D(t) \simeq H'(\tau(t))$.

It is the responsibility of the programmer to select D, an entry point L_O to some functionality of interest in D, H, its target site L_H, and to define $\simeq$ and τ. It is the responsibility of μTrans to find V, O, ϕ, and insert O into H at L_H. An organ in the donor and its version after transplantation are type 4 clones, related by the rewriting system ϕ.

4. THE μTRANS APPROACH

Fig. 2 shows the overall architecture of the μTrans approach to the software transplantation problem. Given an organ's entry point L_O and T_D^O, a test suite that exercises O's behaviour, μTrans progressively 'evolves' O and V by adding

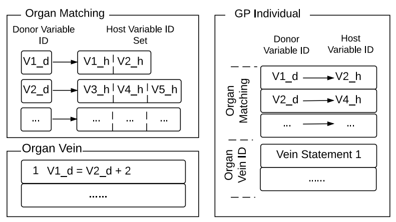


Figure 3: A GP chromosome.

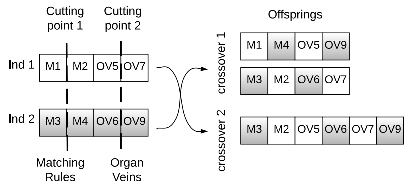


Figure 4: Two crossover operators.

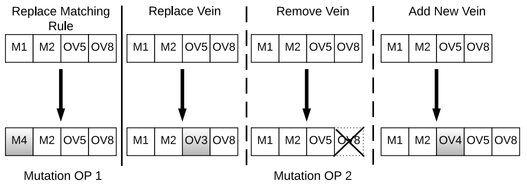


Figure 5: Two mutation operators.

code along a path (a vein) from donor start to L_O. The search space is enormous, consisting of all combinations of all valid statements and variables of the desired functionality in the donor and host. To explore this search space efficiently, we separate our approach into two stages.

In the first, organ-extraction stage, we slice out an over-organ, a conservative over-approximation of the organ and one of its veins. The organ implements the functionality we wish to transplant; the vein builds and initializes an execution environment that the organ expects. In the second, implantation stage, we apply genetic programming, using the donor's test suite, reduce the over-organ and adapt it to execution environment at L_H, the insertion context in the host. The second stage implements observational slicing [8] in GP to simultaneously reduce the over-organ and explore mappings of the organ's parameters to variables in scope at L_H. The second stage ends by implanting the organ into the host. Finally, we apply additional testing to validate the correctness of the postoperative host. This section explains the first and the second stages of μTrans in detail; Sec. 5 discusses its validation.

4.1 Stage 1: Organ Extraction

This stage takes the donor, annotated with L_O, and the organ's entry point in the donor. It produces two outputs for the second stage. First, it constructs an *over-organ*, a vein and an organ, that contains all the code in the donor that implements the organ, given the organ's entry with L_O. Second, it builds the donor's System Dependence Graph (SDG) which is used during GP to reject syntactically invalid offspring.

To produce the over-organ's vein and the organ below, we produce a conservative, over-approximate slice by context-insensitively traversing the donor's call graph: we include all functions called by any function we reach. An organ's vein is a path from a donor's start to L_O, the organ's entry. It contains all source code that constructs the organ's parameters. To create it, we backward slice the donor from L_O, traversing the call graph in reverse. Once we reach the donor's `main` function, we prune the slice to retain only the shortest path, under the assumption that all paths to the organ are equivalent with respect to correctly constructing the organ's parameters. The organ's vein in the case of the running example (Sec. 2) is: '`idct`←`recv_packet`←`transmit_packet`←`send_packet`←`main`'. To construct the initial organ, we forward slice the donor from L_O. The forward slice for the `idct` organ (Sec. 2) is: '`check, dct, malloc, cos, printf`'.

This over-organ is quite imprecise; we rely on the second stage to refine it to a small organ suitable for implantation. Investigating more precise techniques is future work.

Last, this phase outputs data and control dependency information. The GP process may generate many invalid programs, offspring that use undeclared variables or include only a syntactically invalid part of a compound statement, like an `if−else`. Both of these issues occurred in our running example. In `recv.c`, GP selected '`else if (nrOfLostPackets == 2)`' (line 119). Tracking the data dependency ensures we include the declaration '`int nrOfLostPackets`' (line 100), while the control dependency ensures the inclusion of '`if (nrOfLostPackets == 1)`'(line 102), the predicate of the '`else if`' statement, since, in isolation, it violates C syntax.

4.2 Stage 2: Organ Reduction and Adaption

The second stage of μTrans has two steps. The first searches for bindings from the host's variable in scope at the implantation point to the organ's parameters. Some of these variables need to be created and initialized during implantation; others use existing host variables. In the former case, we use α-renaming [2] to avoid variable capture; in the latter, we use types to restrict the space of possible mappings.

We use genetic programming to evolve an organ and its vein from the over-organ produced by the first phase. To this end, we introduce *in-situ* unit testing. Traditional unit testing covers infeasible paths because it does not assume knowledge of how the unit will be used, *i.e.* constraints on its starting program states, and therefore executes paths that are infeasible *in-situ*, when the unit is embedded in a program. *In-situ* unit testing is a novel form of unit testing that starts from a valid program state rather than an arbitrary state. It loops over this state, modifying individuals to maximize coverage.

In-situ is a generally useful concept. In addition to its use in GP, it increased the rigor of our validation. Hosts tend to have large input spaces into which μTrans inserts alien code. Finding a path in the beneficiary to the transplanted organ can be difficult. *In-situ* unit testing allows us to leverage a single path to rigorously test whether the new functionality executes correctly in the host. During *in-situ* testing, any calls the unit makes to its enclosing environment, like its host in autotransplantation, are executed and not stubbed or mocked as they would be in traditional unit testing [4].

μTrans thus realises a form of dynamic, observational slicing which removes redundant states from V while ensuring that O and V remain compilable and executable, and still correctly construct the parameters that O needs to retain the correct behaviour as defined by T_D^O, and *in-situ* unit testing.

Fig. 3 shows an example of the chromosome used in our GP. We inline all functions, then map the over-organ's statements to an array; each array index uniquely identifies each statement. The chromosome of each individual has two parts: a host-to-organ map and a list of the indices in the over-organ that this individual includes.

Unlike conventional GP, which creates an initial population from individuals that contain multiple statements, μSCALPEL generates an initial population of individuals with just 1 statement, uniformly selected. Our underlying assumption is that our organs need very few of the statements in their donor. We also want to evolve small organs. Starting from one LOC gives μSCALPEL the possibility to find small solutions quickly.

Alg. 1 shows the process of generating the initial population. The organ's symbol table stores all feature-related

Algorithm 1 Generate the initial population P; the function `choose` returns an element from a set uniformly at random.

```
Input V, the organ vein; S_D, the donor symbol table; O_M, the host
    type-to-variable map; S_p, the size of population; v, statements in
    individual; m, mappings in individual.
1: $P := \emptyset$
2: for $i := 1$ to $S_p$ do
3:     $m,\ v := \emptyset,\ \emptyset$
4:     for all $s_d \in S_D$ do
5:         $s_h := \text{choose}(O_M[s_d])$
6:         $m := m \cup \{s_d \rightarrow s_h\}$
7:     $v := \{\ \text{choose}(V)\ \}$
8:     $P := P\ \cup\ \{(m, v)\}$
9: return $P$
```

variables used in donor at L_O. For each individual, Alg. 1 first uniformly selects a type compatible binding from the host's variables in scope at the implantation point to each of the organ's parameters. We then uniformly select one statement from the organ, including its vein, and add it to the individual. The GP system records which statements have been selected and favours statements that have not yet been selected.

Search Operators During GP, μTrans applies crossover with a probability of 0.5. We define two custom, crossover operators: fixed-two-points and uniform crossover. The fixed-two-points crossover is the standard fixed-point operator separately applied to the organ's map from host variables to organ parameters and the statement vector, restricted to each vector's centre point. The uniform crossover operator produces only one offspring, whose host to organ map is the crossover of its parents' and whose V and O statements are the union of its parents. The use of union here is novel. Initially, we used conventional fixed-point crossover on organ and vein statements vectors, but convergence was too slow. Adopting union sped convergence, as desired. Fig. 4 shows an example of applying the crossover operators on the two individuals on the left.

After crossover, one of the two mutation operators is applied with a probability of 0.5. The first operator uniformly replaces a binding in the organ's map from host variables to its parameters with a type compatible alternative. In our running example, say an individual currently maps the host variable `curs` to its `N_init` parameter. Since `curs` is not a valid array length in `idct`, the individual fails the organ test suite. Say the remap operator chooses to remap `N_init`. Since its type is `int`, the remap operator selects a new variable from among the `int` variables in scope at the insertion point, which include '`hit_eof`, `curs`, `tos`, `length`, ...' . The correct mapping is `N_list` to `length`; if the remap operator selects it, the resulting individual will be more fit.

The second operator mutates the statements of the organ. First, it uniformly picks t, an offset into the organ's statement list. When adding or replacing, it first uniformly selects a index into the over-organ's statement array. To add, it inserts the selected statement at t in the organ's statement list; to replace, it overwrites the statement at t with the selected statement. In essence, the over-organ defines a large set of addition and replacement operations, one for each unique statement, weighted by the frequency of that statement's appearance in the over-organ. Fig. 5 shows an example of applying μTrans's mutation operators.

At each generation, we select top 10% most fit individuals (*i.e.* elitism) and insert them into the new generation. We use tournament selection to select 60% of the population for reproduction. Parents must be compilable; if the proportion of possible parents is less than 60% of the population, Alg. 1 generates new individuals. At the end of evolution, an organ that passes all the tests is selected uniformly at random and inserted into the host at H_l.

Fitness Function Let I_C be the set of individuals that can be compiled. Let T be the set of unit tests used in GP, TX_i and TP_i be the set of non-crashed tests and passed tests for the individual i respectively. Our fitness function follows:

$$fitness(i) = \begin{cases} \frac{1}{3}\left(1 + \frac{|TX_i|}{|T|} + \frac{|TP_i|}{|T|}\right) & i \in I_C \\ 0 & i \notin I_C \end{cases} \quad (1)$$

For the autotransplantation goal, a viable candidate must, at a minimum, compile. At the other extreme, a successful candidate passes all of the T_D^O, the developer-provided test suite that defines the functionality we seek to transplant. These poles form a continuum. In between fall those individuals who execute tests to termination, even if they fail. Our fitness function therefore contains three equally-weighted fitness components. The first checks whether the individual compiles properly. The second rewards an individual for executing test cases to termination without crashing and last rewards an individual for passing tests in T_D^O.

Implementation Implemented in TXL and C, μSCALPEL realizes μTrans and comprises 28k SLoCs, of which 16k is TXL [17], and 12k is C code. μSCALPEL inherits the limitations of TXL, such as its stack limit which precludes parsing large programs and its default C grammar's inability to properly handle preprocessor directives. As an optimisation we inline all the function calls in the organ. Inlining eases slice reduction, eliminating unneeded parameters, returns and aliasing. For the construction of the call graphs, we use GNU `cflow`, and inherit its limitations related to function pointers.

5. EMPIRICAL STUDY

This section explains the subjects, test suites, and research questions we address in our empirical evaluation of automated code transplantation as realized in our tool, μSCALPEL.

Subjects We transplant code for five donors into three hosts. We used the following criteria to choose these programs. First, they had to be written in C, because μSCALPEL currently operates only on C programs. Second, they had to be *popular* real-world programs people use. Third, they had to be *diverse*. Fourth, the host is the system we seek to augment, so it had to be large and complex to present a significant transplantation challenge, while, fifth, the organ we transplant could come from anywhere, so donors had to reflect a wide range of sizes. To meet these constraints, we perused GitHub, SourceForge, and GNU Savannah in August 2014, restricting our attention to popular C projects in different application domains.

Presented in Tab. 1, our donors include the audio streaming client IDCT, the simple archive utility MYTAR, GNU Cflow (which extracts call graphs from C source code), Webserver[3] (which handles HTTP requests), the command line encryption utility TuxCrypt, and the H.264 codec x264. Our hosts include Pidgin, GNU Cflow (which we use as both a donor and a host), SOX, a cross-platform command line audio processing utility, and VLC, a media player. We use x264 and VLC in our case study in Sec. 7; we use the rest in Sec. 6.

These programs are diverse: their application domains span chat, static analysis, sound processing, audio streaming, archiving, encryption, and a web server. The donors vary in size from 0.4–63k SLoC and the hosts are large, all greater

[3] `https://github.com/Hepia/webserver`.

Table 1: Donor and host corpus for the evaluation.

			Tests	
Subjects	Type	Size	#Regr.	#Unit
Idct	Donor	2.3k	-	3-5
Mytar	Donor	0.4k	-	4
Cflow	Donor	25k	-	6-20
Webserver	Donor	1.7k	-	3
TuxCrypt	Donor	2.7k	-	4-5
x264	Donor	63k	-	5
Pidgin	Host	363k	88	-
Cflow	Host	25k	21	-
SOX	Host	43k	157	-
VLC	Host	422k	20	-

than 20k SLoC. They are popular: Wikipedia reports that more than 3 million people used the Pidgin chat client in 2007. Pidgin, SOX, and TuxCrypt are downloaded from SourceForge over a million times each year on average. VLC, MyTar, and Webserver average 105 forks and 148 watchers on GitHub. x264 is an award winning and GNU Cflow is a well-known, well-maintained static analysis tool.

Test Suites We use three different test suites to evaluate the degree to which a transplantation was successful: 1) the host's pre-existing *regression* test suite, 2) a manually augmented version of the host's regression test suite (*regression++*), and 3) an *acceptance* test suite for the postoperative host, manually updated to test the transplanted functionality at the system-level. A host's developers designed its pre-existing regression test suite and distributed it. Sadly, these suites do not always achieve high statement coverage. We use these test suites to answer **RQ1.1**. To achieve higher statement coverage, we manually added tests to a host's pre-existing test suite to create regression++, our augmented regression suites, for each host. We use these test suites to answer **RQ1.2**. Finally, we manually realized Def. 3, defining input adaption τ and equivalence $\simeq$ for the test oracle, to create an acceptance test suite for the postoperative host: starting from the donor's acceptance tests, one of the authors devised acceptance tests for the postoperative host that execute the transplanted software organ from the postoperative host's entry point. They assess whether the new functionality required is to be found in the host. Our acceptance tests are not weak proxies [49]. Thus, our test cases check whether or not the output of the organ is correct with respect to the original donor program, rather than just checking the exit code. We use these to answer **RQ2**. Our corpus and our test suites are available at `http://crest.cs.ucl.ac.uk/autotransplantation`.

Research Questions Since our approach transplants code from the donor into the host, a natural first question to ask is 'Whether transplantation breaks anything in the host program?'. In biological terms, we are checking the side-effects of the transplantation. In software engineering, this question becomes one of regression testing: 'Does the modified host pass all the regression test cases?'. If it does, then we conclude that we have found no evidence for any side effects of the transplant operation. This motivates our first research question:

Research Question 1: Can we transplant a software organ into host that, after transplantation, still passes all of its regression tests?

Of course, the answer to this question depends critically on the quality of the regression testing. All the programs with which we experimented are real-world systems equipped with test suites, deemed to be useful and practical by their developers. However, they were not designed to test transplants and may not be sufficiently rigorous to find regression faults introduced by transplantation.

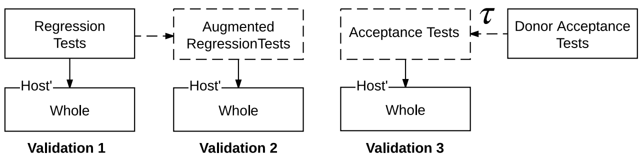


Figure 6: Three validation steps; the dashed boxes are test cases we created.

We computed coverage information for each subject's test suites and found that it was not always high. Finding test suites that achieve high coverage of real-world code using system-level testing is a known challenge. Despite much work on tool development, automating extensive coverage in system-level testing remains an open problem [37]. Furthermore, the value of higher coverage in testing is the subject of ongoing debate in the research community [31].

Nevertheless, since we are injecting new code into the host from a foreign donor, we certainly ought to seek to cover the host system as thoroughly as possible in our testing. We therefore manually augmented each subject's existing regression test suites with additional test cases to increase coverage. Our aim is to more rigorously regression test our transplant operations for side effects. This motivates our two specific versions of RQ1, which focus on each of these coerces of regression test suite:

RQ1.1: Can we transplant a software organ and still pass all of the postoperative host's existing regression tests?
RQ1.2: Can we transplant a software organ and still pass a regression test suite, manually augmented to achieve high coverage in the postoperative host?

Achieving transplantation without side effects is necessary, but not sufficient for success. We need to do more than merely insert alien code into the host without breaking regressions tests; we need to augment the functionality of the postoperative host with new behaviour that replicates the software organ we extracted from the donor, or the operation is pointless. This motivates our second research question:

Research Question 2: Does the transplanted organ provide any new functionality to its postoperative host?

We consider two approaches to test whether the postoperative host exhibits the transplanted functionality. The first, natural question to ask is whether the postoperative host passes acceptance tests, that is, system-level tests that specifically target the new, hoped for behaviour that would represent an implementation of the new feature in the host. For this purpose, we need to adapt the donor's acceptance test suite to be suitable for the postoperative host, *i.e.* define τ in Def. 3. In the most cases, the donor's inputs and the host's are very different. Thus, we have manually defined the τ function, for each transplantation. For example, Cflow takes a set of C source code files as input, while Pidgin is a GUI program. Thus, to supply Cflow with its inputs, after its insertion into Pidgin, we added code to Pidgin that opens a dialog to prompt the user for files. To validate the transplantation, we then compared the output of the Cflow organ in Pidgin against the original Cflow over same set of files.

In summary, our first two research questions draw on three different forms of validation, as outlined in Fig. 6. If we can find transplants that pass regression tests and transfer a meaningful amount of new functionality into the host, satisfying all three of these validation steps, then this would be

Table 2: Transplantation results over 20 repetitions: column *unanimously passing runs* shows the number of runs that passed all test cases in all test suites; column *Test Suites* shows the results for each test suite: *PR* reports the number of passing runs; *All* and *O* report statement coverage (%) for the postoperative host and for the organ; column *Time* shows the execution time (User+Sys) in minutes; * excludes tests that failed due to a pre-existing bug in the organ.

		Unanimously	Test Suites									Time (min)	
			Regression			Regression++			Acceptance				
Donor	Host	**Passing Runs (PR)**	PR	All	O	PR	All	O	PR	All	O	Avg.	Std. Dev.
IDCT	Pidgin	**16***	20	8	0	17	51	99	16*	40	99	5	7
MyTar	Pidgin	**16**	20	8	0	18	51	93	20	40	61	3	1
Web	Pidgin	**0**	20	8	0	0	51	65	18	40	65	8	5
Cflow	Pidgin	**15**	20	8	0	15	52	53	16	41	54	58	16
TuxCrypt	Pidgin	**15**	20	8	0	17	51	88	16	40	88	29	10
IDCT	Cflow	**16**	17	48	91	16	69	91	16	50	99	3	5
MyTar	Cflow	**17**	17	50	52	17	69	90	20	50	91	3	<1
Web	Cflow	**0**	0	3	13	0	61	68	17	49	62	5	2
Cflow	Cflow	**20**	20	71	70	20	71	70	20	71	70	44	9
TuxCrypt	Cflow	**14**	15	46	66	14	69	87	16	50	83	31	11
IDCT	SOX	**15**	18	32	94	17	42	94	16	20	94	12	17
MyTar	SOX	**17**	17	31	62	17	42	90	20	22	60	3	<1
Web	SOX	**0**	0	12	27	0	12	67	17	12	65	7	3
Cflow	SOX	**14**	16	29	47	15	41	66	14	19	59	89	53
TuxCrypt	SOX	**13**	13	35	79	13	41	86	14	20	79	34	13

encouraging evidence that automated code transplantation is a feasible means of extracting and transplanting functionality from donors to hosts. However, there remains the question of the computational cost of this overall approach:

Research Question 3: What is the computation effort to find these organs?

We answer the first 3 questions entirely quantitatively using our study of 15 transplant attempts from 5 donors into 3 hosts including, as a sanity check, one self-application of a donor into itself. With these questions, we give initial empirical evidence to support our claim that μTrans to automated reuse merits further consideration.

However, even if the answers to all of these questions are generally positive and encouraging, there remains the question of whether autotransplantation can deliver *useful* new functionality. This motivates our final research question:

Research Question 4: Can automated transplantation transfer useful new functionality into the host?

To answer this question we use μSCALPEL to automatically transplant a feature into a host that was also developed by human developers. This allows us to qualitatively and quantitatively study an instance of transplantation in which we *know* the goal of transplantation is useful, since humans sought to achieve the same results using traditional manually intensive programming. For this study, we chose the VLC open source media player. The human-implemented feature we choose is the encoder for the H.264 format.

6. RESULTS AND DISCUSSION

For all 15 experiments, we report the number of runs in which all test cases passed in all test suites. We call these *unanimously passing runs* and report them in Tab. 2. We also report the number of successful runs for the regression, augmented regression and acceptance test suites (columns *PR* in Tab. 2). We repeat each experiment 20 times. Before we answer the research questions posed, we summarise our results.

We transplanted an archiving feature from MyTar. Generation of inverse discrete cosine transform coefficients from an array of integers was extracted from the IDCT donor, while the AES encryption feature was extracted from TuxCrypt. We also transplanted our Web donor's functionality that starts a web server and provides file access. Parsing and processing C source code was the feature extracted from Cflow.

Tab. 2 provides evidence that automatic software transplantation is indeed feasible. *In 12 out of 15 experiments all test cases passed*, while 63% (188/300) of all the runs unanimously passed all test suites. *New functionality from 4 out of 5 different donors has been successfully transplanted into the three chosen host systems.* Moreover, within 20 repetitions we were successful in at least 13 runs in these cases. Given that we set ourselves the challenging task of transplanting new functionality into an existing system, μSCALPEL's success rate of at least 65% shows the great promise of μTrans.

Furthermore, our regression failure in the case of the Web donor is that the automatically extracted code contains a loop that listens for server requests. Regression tests were designed for the host prior to transplantation and may not handle nonterminating behaviour. However, in at least 17 runs, the postoperative host passed its acceptance test suites. The Web donor's organ is reachable in the postoperative host and its behaviour retained in all three hosts. When transplanting functionality from Cflow into itself we achieved 100% success rate. This confirms our suspicion that this case would be the easiest for our approach, since issues such as implantation point, variable scope and renaming become less of a concern.

Once we extracted a new functionality, we insert it into the host programs. We ran the test suites provided with Pidgin, Cflow and SOX to check if these software systems retained features deemed important by software developers.

RQ1.1: *Can we transplant a software organ and still pass all given regression tests?* In all but two cases, the beneficiaries pass the regression suites in at least 13 of 20 runs (Tab. 2). Web servers are reactive systems, which contain an event loop. The webserver donor's organ contains such a loop, in which it listens for http requests. Some of Cflow and SOX tests enter the event loop, do not terminate, and fail. *Therefore, the answer to RQ1.1 is that new functionality has been inserted in 13 out of 15 experiments without distorting host program behaviour exercised by the given test suites.* In all transplantation experiments, Pidgin, on the other hand, passed all regression tests. Given this variance, we use `gcov`, a popular coverage tool, to measure the path coverage of our subject's test suites. Tab. 2 reports our results. The *All* columns report coverage rates for the entire host program with new functionality transplanted, while the *O* columns report statement coverage just for the organ. For Pidgin, the statement coverage rate is only 8%, while the transplanted code is not covered at all.

RQ1.2: *Can we transplant a software organ and still pass a regression test suite, manually augmented to achieve high coverage in the postoperative host?* We augmented the hosts' regression test suites with additional tests to increase statement coverage, which we report in Tab. 2. The new test suites now cover 41–71% of the host programs with transplanted features, with one exception, while organ coverage is even higher, exceeding 90% in 6 cases. Furthermore, the success rate was largely retained for Cflow and SOX, where we lost at most one successful passing run for the augmented regression test suite vs. the pre-existing one, as shown in Tab. 2. Coverage rates of Pidgin increased from 8% to 51–52% which had an impact on success rates. The augmented test suite covered 53–99% of the organ's code. Since the organ from Web donor was not covered by the original regression test suite and the organ contains an infinite loop, as pointed out above, the augmented suite has a pass rate of 0%. *Therefore, our answer to RQ1.2 is that postoperative hosts passed all of their augmented regression tests in 12 out of 15 experiments.*

RQ2: *Does the transplanted organ provide any new functionality to its postoperative host?* Next, we test whether the organ is reachable within the host and if it retains the desired behaviour. The numbers of successful runs of these acceptance test suites, and the statement coverage are presented in Tab. 2. The statement coverage ranges from 54 to 99% for the organs and 12 to 71% for the whole host program with transplanted new functionality. Since these tests are aimed at the organ code, we did not expect to get high coverage in the hosts, which is the lowest, 12–22%, in the case of SOX. We suspect that dead code, from older audio formats that are no longer supported, is the reason. However, organ coverage always exceeds 59%. The highest coverage was achieved for Cflow, which contains recursive calls.

The number of times the entire acceptance suite passed ranges from 14 to 20. This means that new features are indeed found in the hosts in at least 70% of the repeated runs. Moreover, all subjects with new functionality from MyTar pass all acceptance tests in all 20 runs, hence the organ is always reachable. Furthermore, by transplanting a piece of IDCT into Pidgin, we discovered a bug in the original code, not reachable in the donor. With Pidgin as its host, however, the IDCT organ can be reached with an empty array which the code does not handle. The success rates presented in Tab. 2 are based on the number of tests passed excluding our test cases that discover the bug, since it is present in the donor and we measure faithfulness to the original program. Therefore, not only we have shown that we can transplant code from one program into another, but also these new features will be reachable and executable. *To answer RQ2, in all 15 experiments the transplanted features are accessible and provide the desired functionality. Moreover, 85% (256/300) of the runs passed all acceptance tests.*

RQ3: *What is the computation effort to find these organs?* We have shown that we can successfully transplant new functionality into an existing software system. The question remains, how efficient is our approach? Average timings of extracting and transplanting a new feature into the host program are shown in Tab. 2 (column *Avg*). *To answer RQ3, average runtime of one run of the transplantation process did not exceed 1.5 hour and in 8 cases it was less than 10 minutes.* This shows the efficiency of our approach. If you consider the time it would have required for a human to perform the same tasks of adding new functionality, we believe that these timings would have been in the order of hours or days rather than minutes.

7. A CASE STUDY

Despite the fact that the answers to research questions 1–3 in Sec. 5 are generally positive and encouraging, the question of whether automatic transplantation can deliver *useful* new functionality remains: **RQ4:** *Can automated transplantation transfer useful, new functionality into the host?* We answer this question by considering the automated transplantation of a feature into a host that human developers implemented. This allows us to qualitatively and quantitatively study the transplantation of a feature whose transplantation we *know* to be useful, since humans worked to achieve the same results using traditional, manually intensive programming.

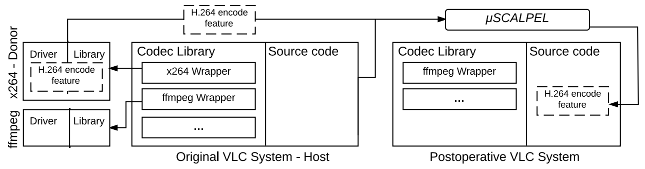


Figure 7: μSCALPEL autotransplants x264's encoder *functionality* into VLC; the donor is x264 , the organ its H.264 encoder, and the host is VLC, stripped of any reference to x264 prior to transplantation.

VLC is a popular open source media player[4]; VLC seeks to offer a versatile media player that 'can play any file you can find'. To achieve this goal, VLC is under continual development, especially to handle new media formats. x264 is a popular, widely-used, award winning, open source video and audio encoding tool[5], designed for the H.264/MPEG-4 AVC compression format[6]. Both programs are substantial: at the time of this writing, x264 contains more than 63k lines of code and 211 files; version 2.1.5 of VLC contains more than 422k lines of code across 2821 files.

H.264 is an evolving video standard, to which features are continually added. Keeping a code base current with an evolving standard like H.264 is time-consuming. Currently the developers of VLC manually update the code related to x264 library, when its interface changes. VLC has invested considerable effort in supporting H.264, first implementing it in November 2003. Since then, over 450 commits in VLC's version history mention H.264. These commits run the gamut, including bug fixes, new features, or refactoring. The most recent change to mention H.264 was committed 21 January 2015. VLC's developers have worked over 11 years at maintaining and updating the handling of the H.264 format.

To insulate themselves from changes in x264, the VLC community includes it as a library veiled by wrapper, which they must update when it changes. Autotransplantation can sped this process by automatically handling renaming and, as we describe below, it also takes only that part of the x264 codebase that VLC actually uses. To use μSCALPEL here, a VLC developer strips out all the code at an implantation point (conceptually, this removes the previous version) as μSCALPEL implants an organ as an atomic entity, it does not merge it into existing code.

Consider Fig. 7. At the left is the original VLC system, which links all of x264 as an external library and interacts with its encoder through a wrapper. At the right is the host, which we produced from original VLC codebase by stripping all references to x264, by building it using `./configure −−enable−x264=no`; this setting does not link the x264 library and eliminates the wrapper via preprocessor directives. This step ensures that μSCALPEL transplanted new, rather than uncovering old, functionality.

We marked `x264_encoder_close` (`encoder.c`:2815) as the organ entry point and defined an organ test suite by using the original x264 implementation as the oracle. In the host, we annotated `input_DecoderNew` (`decoder.c`:314) as the implantation point, because VLC calls this method for encoding or streaming a video, and choosing what encoder to use.

Using the organ entry point annotation and the organ test suite, μSCALPEL extracts the encoder organ, the box labeled

[4]`http://www.videolan.org/vlc`.
[5]`http://www.videolan.org/developers/x264.html`.
[6]`http://en.wikipedia.org/wiki/H.264/MPEG-4_AVC`.

'H.264 encode feature' in Fig. 7, from the donor, then implants it into the stripped version of VLC at the designated implantation point to produce a postoperative version that includes x264 encoder functionality. Due to its use of slicing, μSCALPEL takes only what VLC needs and uses from x264, unlike the human process which replaces the x264 as a library and updates its wrapper.

7.1 Study Design and Setup

x264 is multi-threaded, and so is the organ that μSCALPEL extracted from it and transplanted. We transplanted the H.264 organ from x264 with the assembly optimisations disabled, because μSCALPEL is built on TXL's C grammar, which does not handle assembly code. The latest version of VLC (2.1.5) was our target host, after manually disabling its existing H.264 functionality by building it with `./configure −−enable−x264=no`. As noted above, the implantation point is `input_DecoderNew`, VLC's encoder selection method.

Before we could apply μSCALPEL to x264, we had to manually workaround TXL's limitations in handling C preprocessor directives: by default, TXL treats preprocessor directives as strings. Programmatically resolving an `#ifdef` as a string requires replacing it with its `then` branch or its `else` branch or both. In any nontrivial program, like x264, any of these choices either fail to compile or generate invalid code.

For our evaluation, we chose 5 movie trailers, with lengths varying between 10 and 138 seconds, frames between 150 and 2260, and bit rates between 3218 and 9419. We ran both of our versions of VLC — original and with the transplant — in command line mode, bypassing its GUI. We used VLC's command line parameter `vlc://quit` to close VLC when a video ends. We used the same encoding options across all videos and all program versions: x264 without and with x264 ASM optimisations, original VLC, and VLC with the x264 organ implanted. These options are '4:4:4 Predictive' for the profile option (colour handling), 4:2:0 for YUV (luminance and chrominance weight), and 2.2 for level (maximum bit rate).

7.2 Observations

We transplanted a specific version of x264 into VLC. We cannot precisely measure how much time the VLC community spent updating their code for new versions of x264, so we bound it using VLC's version history. Over 11 years, the VLC community has upgraded VLC many times. We assumed that these upgrades triggered a burst of commits, where we consider a burst to be sets of four commits no more than 10 days apart. In VLC's version history, we identified 39 commit bursts whose logs contain '264'. On average, an H.264 commit occurred every 9 days. The average number of commits across the 39 bursts is 8; 312 commits were in bursts, while 126 were isolated. Each commit burst spanned 20 days on average. We uniformly selected the commit burst starting 5 January 2012 and ending 21 January 2012. It has 12 commits. Its commits changed 2 files. The `diffstat` of these commits is 171.

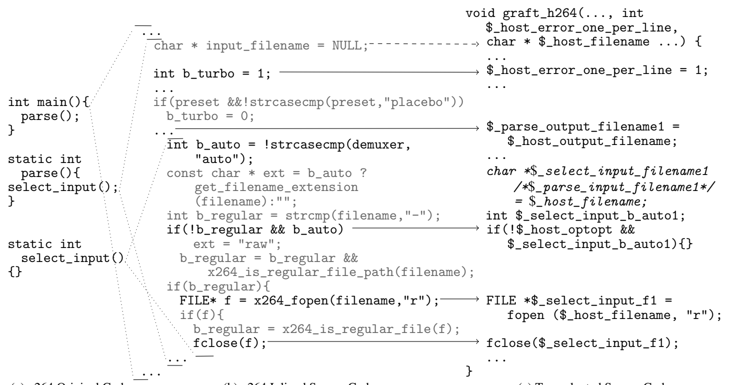
The total size of the organ μSCALPEL extracted was 23k LOCs, including the veins, as reported by `cloc`. In contrast, an over-approximate estimate of the equivalent human organ is 12k in VLC and 44k SLoC in its x264 library. Half of the μSCALPEL's organ's lines are global variable declarations; type and function definitions consume half the lines; inlined function calls are the next largest source of lines. μSCALPEL performs standard renaming, but is also able to reuse declarations in the host. For example, both VLC and x264 contain the typedef `uint8_t`. In the vein, we have '`uint8_t output_csp_fix[]`' ; in the host, we have '`uint8_t *p`' in scope. Since the types are compatible, μSCALPEL unifies them, and removes `uint8_t` from the organ, since it is no longer needed. The organ likely contains unnecessary functions. In the future, one could post-process μSCALPEL's output to shrink the organ, as proposed in GI work [41, 48].

Fig. 8 shows μSCALPEL in action. On the left, Fig. 8a contains a snippet of code as it appears in the donor, after slicing has identified it as part of the vein in the donor. Fig. 8b shows the organ after extraction into the ice-box, with all the functions in the vein inlined. The dotted lines bracket the regions inlined from Fig. 8a to Fig. 8b. At the right, Fig. 8c shows the organ after transplantation into the host. From Fig. 8b to Fig. 8c, the dashed lines show deletions and the solid lines show variable equivalences. At the right, we see `graft_h264`, the entry point into the organ in the host. Its parameter list is all variables in scope at the insertion point in the host. The organ may not use all of these parameters. At entry, `graft_h264` hooks the host variable (as parameters) to the organ's variables, then executes the code from the organ's vein to initialize state before calling the organ. This is why this snippet has many deletions and is dominated by assignment statements. For example, the organ reads the video stream from `$_host_filename`. We use TXL's built facility for generating fresh names to avoid variable capture to generate the names you see. As you can see, μSCALPEL reduces an `if` statement to a NOP; organ minimization is future work.

Fig. 8 focuses on initialization code; Fig. 9 shows how μSCALPEL modified functional organ code during autotransplantation. x264 heavily uses `x264_malloc`, its wrapper for the standard `malloc` function. μSCALPEL specialises it for different calling contexts. Here, x264 has called it to allocate memory for `x264_t`, which is bigger than '`2 * 1024 * 1024 * 7 / 8`' . This allocation always succeeded in our test cases, so μSCALPEL eliminated the unneeded `if` statements, as shown. Since function calls can occur in arbitrary expression and μSCALPEL greedily inlines their definition upon encountering them, it replaces return statements with assignment to a fresh variable. To skip any code that follows the replaced return, whose execution the return blocked, μSCALPEL follows the assignment with a GOTO to a label after the call in the caller (2 in Fig. 9). When a declaration follows the added LABEL, as with void functions, μSCALPEL generates a NOP to preserve syntactic correctness, without affecting semantics.

μSCALPEL needed only a single run and 26 hours to extract and transplant x264's H.264 encoder into VLC; in other words, it passed all the regression, augmented regression and acceptance test suites. Since statement coverage for VLC's existing test suite was 11.5% and 0% for the organ, we manually added tests. The resulting, augmented test suite combined with our acceptance tests achieved 63% coverage of the beneficiary and 49.4% of the organ. After 50 runs over our test cases (10 for each video trailer), `gcov` reported a coverage of 49.4%.

As mentioned, μSCALPEL works just on C. In the case of video encoding, ASM optimisations vastly improve encoding speed. Because of this limitation, our runtime results are up to 8 times slower than those of the original VLC program. Here again, we can, in the future, turn to GP to optimise efficiency of the beneficiary: a recent technique speed up existing code 70-fold [41, 48]. We also measured the size of the encoded video. The output of our transplanted organ matches the size of its output when run in the donor, in



```
int main(){
  parse();
}

static int
  parse(){
select_input();
}

static int
  select_input()
{}
```

(a) x264 Original Code

```
...
char * input_filename = NULL;

int b_turbo = 1;
...
if(preset &&!strcasecmp(preset,"placebo"))
  b_turbo = 0;
...
  int b_auto = !strcasecmp(demuxer,
      "auto");
  const char * ext = b_auto ?
      get_filename_extension
      (filename):"";
  int b_regular = strcmp(filename,"-");
  if(!b_regular && b_auto)
      ext = "raw";
    b_regular = b_regular &&
        x264_is_regular_file_path(filename);
  if(b_regular){
    FILE* f = x264_fopen(filename,"r");
    if(f){
      b_regular = x264_is_regular_file(f);
      fclose(f);
  ...
...
```

(b) x264 Inlined Source Code

```
void graft_h264(..., int
    $_host_error_one_per_line,
    char * $_host_filename ...) {
    ...
    $_host_error_one_per_line = 1;
    ...

    $_parse_output_filename1 =
       $_host_output_filename;
    ...
    char *$_select_input_filename1
        /*$_parse_input_filename1*/
        = $_host_filename;
    int $_select_input_b_auto1;
    if(!$_host_optopt &&
       $_select_input_b_auto1){}

    FILE *$_select_input_f1 =
       fopen ($_host_filename, "r");

    fclose($_select_input_f1);
    ...
}
```

(c) Transplanted Source Code

Figure 8: Code snippet from the beginning of the x264 organ: ⇢| is function inlining; ⇢ is variable binding; → is α–renaming; grayed lines are deleted. The last if was dropped because the organ is unreachable with an empty file in VLC.

```
I  <<h = x264_encoder_open_142 (param);>>
+  x264_param_t *$_x264_encoder_open_142_param1 = $_encode_param1;
+  x264_t *$ABSTRETVAL_ret_x264_encoder_open_1421;
α  x264_t *$_x264_encoder_open_142_h1;
    do {
        do {
I        <<x264_malloc (sizeof (x264_t))>>
+        $_host_error_one_per_line = sizeof (x264_t);
/        uint8_t *$_x264_malloc_align_buf2;
/        $_x264_malloc_align_buf2 = ((void *) 0);
—         if (i_size >= 2 * 1024 * 1024 * 7 / 8) {
α           $_x264_malloc_align_buf2 = memalign (2 *1024 *
     1024,$_host_error_one_per_line);
—             if (align_buf) {
α              size_t $_x264_malloc_madv_size2 = ($_host_error_one_per_line + ...);
α              madvise ($_x264_malloc_align_buf2, $_x264_malloc_madv_size2, 14); }
—             else align_buf = memalign (32, i_size);
     ...
—      return $_x264_encoder_open_142_h1;
+     $ABSTRETVAL_ret_x264_encoder_open_1421 = $_x264_encoder_open_142_h1;
+     goto LABEL_x264_encoder_open_142_1;
     ...
—   return ((void *) 0);
+  $ABSTRETVAL_ret_x264_encoder_open_1421 = ((void *) 0);
+  LABEL_x264_encoder_open_142_1:
+  $_encode_h1 = $ABSTRETVAL_ret_x264_encoder_open_1421;
```

Figure 9: Standard diff of code transplanted from the 'library' part of x264, augmented with: inlining('I'); α–renaming ('α'); initialisation separation ('/'). μSCALPEL has specialised the function x264_malloc; for the host, the predicate of the first if is always true, so GP removed it.

every case. However, the size of the output of our organ after transplantation is up to 1.7 times as big as in the original VLC. This is because we provided μSCALPEL with test cases that tested x264's lossless encoding, which μSCALPEL therefore extracted into the organ, not x264's lossy encoding; VLC, in contrast, uses lossy encoding. *To answer RQ4, our tool needed just 26 hours for extracting and transplanting the H.264 multi-threaded encoding feature from x264 into VLC.*

8. RELATED WORK

Our transplantation approach builds on recent work in Genetic Improvement (GI) [1, 27, 41, 43, 47, 56, 64], which treats code as 'genetic material' to be manipulated to improve systems. GI can repair broken functionality [1, 43, 56], dramatically scale-up performance [40–42, 47, 52, 54, 64], and port between languages and platforms [39]. It is a kind of program synthesis [45] an area that has recently been the subject of much activity [21].

The idea of software transplantation as a form of Genetic Improvement was recently introduced by Harman et al. [28]. Previously, Weimer et al. [61] had transplanted code from one location to another within the same system for automated repair, while more recently, Petke et al. [48] transplanted fragments of code from multiple versions of the same system for performance improvement. However, the present paper transplants functionality between *different* systems. We believe it is the first to do this, although previous work [26] has grafted new functionality (grown from scratch) into existing systems and, independently (but at the same time as our work), Sidiroglou-Douskos [51] used transplantation for automated repair. Miles at al. [46] reused in-situ logically extracted functionalities from a running program, by using a debugger and manual annotations.

In order to transplant code from a donor to an unrelated host, we must capture the code upon which the chosen functionality depends in the donor, a problem closely related to program slicing [7, 10, 25, 53, 57, 62]. Slices can be constructed in a backward or forward direction [30]. Our approach most closely resembles backward slicing. Slices can be static [30, 63], dynamic [34] and various flavours in between [6, 11, 20, 22, 59]. Our approach is closer to dynamic slicing, but it is guided by test suite observation [8, 9], rather than dependence analysis, and with only a limited form of control dependence [19]. Our slices also need only capture the particular features of interest and not the entire computation on the slicing criterion, thereby resembling 'barrier' slicing [35] and feature extraction [18, 24]. The slices required by transplantation are transformed, making them amorphous [23]. Within the (considerably rich and diverse) slicing nomenclature, our slices can therefore regarded as 'amorphous observation-based dynamic backward barrier slices'.

Ray et al. investigate the problem of keeping a transplanted organ in sync with its clones, as the organs change over time [50]. In their setting, an organ is manually moved to a host. When both original organ and its clone change, the SPA tool checks the two patches for consistency, and flags the inconsistencies if found. In our work we automate the transplantation of functionality from a donor to a host.

9. CONCLUSION AND FUTURE WORK

In this paper, we have introduced μTrans, an approach to autotransplantation that combines static and dynamic analysis to extract, modify, and transplant code from a donor system into a host. We have sought to migrate functionality from one system into an entirely different system, and validate that the result passes regression testing and acceptance testing of the new functionality.

Autotransplantation is a new (and challenging) problem for software engineering research, so we did not expect that all transplantation attempts would succeed. We systematically autotransplanted five donors into three hosts. 12 out of these 15 experiments contain at least one successful run. μSCALPEL successfully autotransplanted 4 out of 5 features, and all 5 passed acceptance tests. In all, 65% (188/300) runs pass all tests in all test suites, giving 63% unanimous pass rate; considering only acceptance tests, the success rate jumps to 85% (256/300).

Our case study compared the autotransplantation of adding support for a new media format to the VLC media player, a task humans had previously accomplished. This study indicates that automated transplantation can be useful, since we showed μSCALPEL could insert support for H.264 encoding into the VLC media player in 26 hours, passing all regression and acceptance tests. Our evaluation provides evidence to support the claim that automated transplantation is a feasible and, indeed, promising new research direction. You can download μSCALPEL from `http://crest.cs.ucl.ac.uk/autotransplantation` and join us in its exploration.

10. REFERENCES

[1] A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *CEC*, pages 162–168, 2008.

[2] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[3] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15th *Conference on Software Engineering (ICSE'93)*, pages 509–518, Los Alamitos, California, USA, 1993. IEEE Computer Society Press.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[6] A. Beszédes and T. Gyimóthy. Union slices for the approximation of the precise slice. In *IEEE International Conference on Software Maintenance*, pages 12–20, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[7] D. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[8] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In 22nd *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, pages 109–120, Hong Kong, China, November 2014.

[9] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo. Observation-based slicing. Technical Report RN/13/13, Computer Sciences Department, University College London (UCL), UK, June 20th 2013.

[10] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[11] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

[12] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance*, pages 424–433, Los Alamitos, California, USA, Sept. 1994. IEEE Computer Society Press.

[13] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Decomposing legacy programs: A first step towards migrating to client–server platforms. In 6th *IEEE International Workshop on Program Comprehension*, pages 136–144, Los Alamitos, California, USA, June 1998. IEEE Computer Society Press.

[14] G. Canfora, A. De Lucia, and M. Munro. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, 1998.

[15] G. Canfora and M. Di Penta. New frontiers in reverse engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, pages 326–341, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[16] A. Cimitile, A. R. Fasolino, and P. Marascea. Reuse reengineering and validation via concept assignment. In *International Conference on Software Maintenance (ICSE 1993)*, pages 216–225. IEEE Computer Society Press, Sept. 1993.

[17] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[18] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003. Special issue on ICSM 2001.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[20] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

[21] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 46(1):317–330, Jan. 2011.

[22] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, Mar. 1995.

[23] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[24] M. Harman, N. Gold, R. M. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 11 – 21, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[25] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[26] M. Harman, W. B. Langdon, and Y. Jia. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In 6th *Symposium on Search Based Software Engineering (SSBSE 2014)*, pages 247–252, Fortaleza, Brazil, August 2014. Springer LNCS.

[27] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In 27th *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 1–14, Essen, Germany, September 2012.

[28] M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for reverse engineering (keynote paper). In R. Oliveto and R. Robbes, editors, 20th *Working Conference on Reverse Engineering (WCRE 2013)*, pages 1–10, Koblenz, Germany, 14-17 October 2013. IEEE.

[29] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In 20th *International Conference on Software Engineering (ICSE '98)*, pages 74–83. IEEE Computer Society Press, Apr. 1998.

[30] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[31] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[32] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[33] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In 27th *Symposium on Principles of Programming Languages (POPL-00)*, pages 155–169, N.Y., Jan. 19–21 2000. ACM Press.

[34] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[35] J. Krinke. Barrier slicing and chopping. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 81–87, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[36] J. Krinke. Is cloned code older than non-cloned code? In J. R. Cordy, K. Inoue, S. Jarzabek, and R. Koschke, editors, 5th *ICSE International Workshop on Software Clones, IWSC 2011*, pages 28–33, Waikiki, Honolulu, HI, USA, 2011. ACM.

[37] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In 4th *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[38] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[39] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In P. Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18-23 July 2010. IEEE.

[40] W. B. Langdon and M. Harman. Genetically improved CUDA C++ software. In 17th *European Conference on Genetic Programming (EuroGP)*, pages 87–99, Granada, Spain, April 2014. Springer.

[41] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEVC)*, pages 118–135, 2014.

[42] W. B. Langdon, M. Modat, J. Petke, and M. Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 951–958, New York, NY, USA, 2014. ACM.

[43] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[44] F. Lianubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[45] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–164, 1971.

[46] C. Miles, A. Lakhotia, and A. Walenstein. In situ reuse of logically extracted functional components. *Journal in Computer Virology*, 8(3):73–84, 2012.

[47] M. Orlov and M. Sipper. Flight of the FINCH through the java wilderness. *IEEE Transactions Evolutionary Computation*, 15(2):166–182, 2011.

[48] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In 17th *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014. To Appear.

[49] Z. Qi, F. Long, S. Achour, and M. Rinard. Efficient automatic patch generation and defect identification in kali. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2015. To appear.

[50] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Automated Software Engineering (ASE 2013))*, pages 367–377, Palo Alto, California, 2013.

[51] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, P. Piselli, and M. Rinard. Automatic error elimination by multi-application code transfer. In 36nd *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, Portland, Oregon, June 2015. To appear.

[52] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, pages 124–134, 2011.

[53] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12:1 – 12:48, June 2012.

[54] P. Sitthi-amorn, N. Modly, W. Weimer, and J. Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152:1–152:11, 2011.

[55] J. E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[56] J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-o-fix: An embeddable framework for dynamic adaptive genetic improvement programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, 2014.

[57] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[58] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. Technical Report SEN-R9814, Centrum voor Wiskunde en Informatica (CWI), Sept. 1998.

[59] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[60] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 455–465, Saint Petersburg, Russian Federation, August 2013. ACM.

[61] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver, Canada, 2009.

[62] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[63] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[64] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation (TEVC)*, 15(4):515–538, 2011.