CodeCarbonCopy

Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, Martin Rinard
stelios,elahtinen,aeden,fanl,rinard@csail.mit.edu
MIT EECS and MIT CSAIL, Cambridge, MA, USA

ABSTRACT

We present CodeCarbonCopy (CCC), a system for transferring code from a donor application into a recipient application. CCC starts with functionality identified by the developer to transfer into an insertion point (again identified by the developer) in the recipient. CCC uses paired executions of the donor and recipient on the same input file to obtain a translation between the data representation and name space of the recipient and the data representation and name space of the donor. It also implements a static analysis that identifies and removes irrelevant functionality useful in the donor but not in the recipient. We evaluate CCC on eight transfers between six applications. Our results show that CCC can successfully transfer donor functionality into recipient applications.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Reusability**;

KEYWORDS

Automatic code transfer

ACM Reference format:
Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106269

1 INTRODUCTION

Software developers often transfer functionality between applications by copying code originally developed for one application (the donor application) into another application (the recipient application). Current practice involves manually adapting the copied code to operate within the environment of the recipient. The software development overhead associated with this manual adaptation can complicate the ability of developers to rapidly and easily transfer functionality between applications. One particularly challenging aspect of transferring code between applications is translating the data representation from the donor representation to the recipient representation.

1.1 CodeCarbonCopy (CCC)

We present CodeCarbonCopy (CCC):
Functionality Identification: The developer identifies the *donor function* that implements the functionality to transfer. The code that implements this functionality is the *transferred code.*
Insertion Point: The developer identifies a program point in the recipient as the *insertion point* of the transferred code.
Donor and Recipient Executions: CCC executes both the donor and the recipient on the same *seed input file* (CCC is designed to support code transfer between applications that can process the same input file format). The executions are instrumented to record symbolic expressions for every value that the applications compute. These expressions record the complete computation of the value as a function of the constants and input file bytes that contribute to its value. CCC uses the symbolic expressions to compute a mapping from the recipient data representation to the donor data representation.
Input and Output Adapters: The transferred code accesses data in the original donor data representation. To enable the transferred code to execute in the recipient environment, CCC uses the mapping to automatically generate *input* and *output* data adapters. The input data adapter reads the recipient data structures to build and populate the donor data structures that the transferred code will access when it executes. The output adapter writes the computed values back from the donor data structures into the recipient data structures.

The CCC adapters work with forests of structures, where the leaves can be arrays of primitive types such as integers or characters. The arrays can be dynamically allocated and the sizes of the arrays can depend on the input. The CCC data structure mappings are based on *matched arithmetic sequences*, which map data indexed by one arithmetic sequence in the recipient to data indexed by another arithmetic sequence in the donor. For example, CCC can use matched arithmetic sequences to translate data stored in separate arrays in one application into data stored interleaved in a single array in another application and vice-versa (Section 3).
Irrelevant Functionality Removal: CCC deploys a static analysis that identifies and removes *irrelevant code* (e.g., GUI code) that accesses *irrelevant values* derived from developer-identified irrelevant parameters to the transferred code.
Code Extraction and Transfer: CCC extracts the transferred code from the donor into the insertion point in the recipient. To facilitate the extraction, CCC transforms global variable references in the transferred code into parameter references. The generated code at the insertion point 1) executes the input adapter to build and populate the donor data structures, 2) invokes the donor function (which executes on the donor data structures), and 3) executes the output adapter to write the computed values from the donor data structures back into the recipient data structures.
Experimental Results: We evaluate CCC on eight transfers between six applications: VLC 2.0.8, MPlayer svn34540, cwebp 0.3.1,

bmp2tiff 4.0.3, ViewNior-1.4 and mtPaint 3.40. CCC was able to successfully transfer functionality for seven out eight transfers. The transfers ranged in functionality from image manipulation transformations to supporting an entirely new image format (CCC retrofitted mtpaint with the ability to read Google's WebP format).

1.2 Contributions

- **Data Representation Translation:** It presents a novel data representation translation technique that automatically translates source-level recipient naming and data representation into source-level donor naming and data representation. The translation is driven by instrumentation that computes application-independent representations of values that the applications compute. The data representation translation for dynamic arrays is driven by inferred *matched arithmetic sequences*, which implement a variety of mappings from recipient to donor arrays.
- **Code Extractor:** It presents a novel code extractor that transitively identifies and extracts all code required to implement the transferred functionality. The extractor also lifts all accessed global variables into parameters of the transferred code.
- **Irrelevant Functionality Removal:** It presents a novel static analysis that automatically identifies and removes code that implements irrelevant functionality.
- **Code Transfer:** It shows how to leverage data representation translation and irrelevant functionality removal to obtain an effective source-level code transfer mechanism.

2 EXAMPLE

We next present an example that illustrates how CCC automatically transfers image rotation functionality from mtpaint, an image processing application [4], to cwebp, an image converter for Google's WebP format [2].

Functionality Identification: The developer identifies the functionality to transfer by specifying the *donor function* that implements that functionality. Here the developer specifies the `mem_sel_rot` function, which rotates the image stored in the `mem_clip.img[0-1]` global data structure by a direction specified by the single parameter `dir`:

```c
int mem_sel_rot(int dir) {// Rotate clipboard 90 degrees
  unsigned char *buf = NULL;
  int i, j = mem_clip_w * mem_clip_h, bpp = mem_clip_bpp;
  for (i = 0; i < NUM_CHANNELS; i++ , bpp = 1) {
    if (!mem_clip.img[i]) continue;
    buf = malloc(j * bpp);
    if (!buf) break;  // Not enough memory
    mem_rotate(buf, mem_clip.img[i], mem_clip_w,
     mem_clip_h, dir, bpp);
    free(mem_clip.img[i]);
    mem_clip.img[i] = buf;
  }
  /* Don't leave mix of rotated and unrotated channels*/
  if (!buf && i) mem_free_image(&mem_clip, FREE_ALL);
  if (!buf) return (1);
  i = mem_clip_w;
  mem_clip_w = mem_clip_h;     // Flip geometry
  mem_clip_h = i;
  return (0);}
```

Listing 1: Donor Functionality: mem_sel_rot

The `mem_sel_rot` function calls the `mem_rotate` function (Listing 2) once for each channel (i.e., BGR, alpha), with with `bpp` values of 3 (for the BGR channel) and 1 (for the alpha channel). For each call to `mem_rotate`, `mem_sel_rot` allocates a buffer `buf` of size `mem_clip_w` * `mem_clip_h` * `bpp` to hold the rotated data for that channel (note that `mem_clip_w`, `mem_clip_h`, and `mem_clip_bpp` are macros for `mem_clip.width`, `mem_clip.height`, and `mem_clip.bpp`). `mem_rotate` invokes GUI code to reflect progress in the application title bar (lines 4 and 6). This GUI code is irrelevant functionality that CCC will eventually delete before the transfer.

```c
1 void mem_rotate( char *new, char *old, int old_w, int
      old_h, int dir, int bpp )
2 {
3   ...
4   if (flag) progress_init(_("Rotating"), 1);
5   ...
6   if (flag) progress_end();
7 }
```

Listing 2: Donor Functionality: mem_rotate

Insertion Point Identification: The developer next identifies the insertion point in cwebp. This insertion point is located after cwebp has decoded the PNG input file into an intermediate ARGB format buffer and before the ARGB buffer is used to output a WebP image.

Donor and Recipient Executions: CCC executes instrumented versions of the donor and recipient on the same input file (a PNG file). The instrumentation dynamically tracks the flow of input bytes through the program to obtain symbolic expressions, in terms of constants and input bytes, for every value that the applications compute.

When execution reaches the donor function or the recipient insertion point, the instrumentation uses the debugging information in the executable to identify the parameters, local variables, and global variables available at that point. Starting with these variables as roots, it traverses the data structures to find source-level expressions (in the name space of the donor or recipient) for accessible values. This traversal generates a log file that contains a symbolic representation of the accessible (via parameters, local variables, or global variables) state in the donor or recipient at that point. Each entry in the log file records a source-level expression (in the donor or recipient name space), the value to which the expression evaluates, and the recorded symbolic expression for the value.

CCC turns off recipient instrumentation after execution reaches the insertion point. During the execution of the donor function, the instrumentation generates log file entries that record each executed instruction and the source-level expression, concrete value, and symbolic expression for every value that the instruction accesses. CCC turns off donor instrumentation when the donor function completes.

In our example, the donor log file contains entries for `mem_clip.width`, `mem_clip.height`, `mem_clip.bpp`, and `mem_clip.img[0-1]`. These variables store the width, height, image bits per pixel, and image data in the donor. The recipient log file contains entries for `picture.width`, `picture.height`, and `picture.argb` (which holds the image data in the recipient). The symbolic expression for both `mem_clip.width` at the start of the donor function and `picture.width` at the insertion point is `ToSize(32,BvAnd(16,OffsetVar('0',26,27),Constant(16383)))`.

In And InOut State Identification: CCC uses the log file from the execution of the donor function to identify In state (which the donor reads but does not write), InOut state (which the donor reads before writing), and Out state (which the donor writes before reading). CCC identifies `dir` and `mem_clip.bpp` as In state and `mem_clip.width`, `mem_clip.height`, and `mem_clip.img[0-1]` as InOut state (there is no Out state in the example). CCC identifies `mem_clip.width` and `mem_clip.height` as InOut state because they are flipped to implement the changes in the image geometry (Listing 1, lines 20-22).

Data Representation Translation: CCC uses the symbolic information in the log file to drive the mapping between the In and InOut donor state and the recipient state. For scalar variables and static arrays, CCC uses the CodePhage rewrite algorithm ([18], Figure 7) to map the symbolic expressions from the recipient into the name space of the donor. This algorithm uses an SMT solver (Z3) to find equivalent expressions that evaluate to the same value. In our example, CCC determines that `picture.width` and `picture.height` in cwebp (recipient) map to `mem_clip.width` and `mem_clip.height` in mtpaint (donor). Because `bpp` is not computed from input file bytes, it is not mapped.

The mapping from `picture.argb` to `mem_clip.img[0-1]` is more complex. cwebp (the recipient) stores image data in a single data structure in the form (B_0,G_0,R_0,A_0, B_1,G_1,R_1,A_1, B_2,G_2,R_2,A_2,...). mtpaint (the donor), on the other hand, stores image data as channel separated BGR (`mem_clip.img[0]`) and alpha (`mem_clip.img[1]`) channels. In other words, mtpaint stores the image RGB channel in the following form: (R_0, G_0, B_0, R_1, G_1, B_1, R_2, G_2, B_2,...) and the alpha channel separately: (A_0, A_1, A_2,...).

We note that the recipient and donor data structures do not typically store data in the same format as the input file — the data arrives at these data structures only after traveling through multiple layers of both library and application input processing code. Typical applied transformations include endian conversions and application-level data reorganization and reformatting. This is the case in our example.

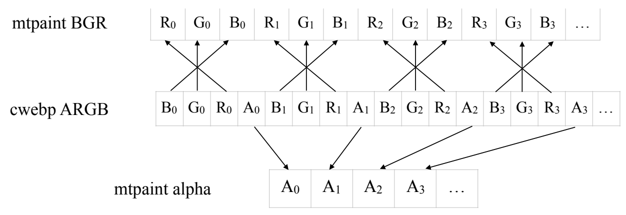
mtpaint BGR
R0 G0 B0 R1 G1 B1 R2 G2 B2 R3 G3 B3 …
cwebp ARGB
B0 G0 R0 A0 B1 G1 R1 A1 B2 G2 R2 A2 B3 G3 R3 A3 …
mtpaint alpha
A0 A1 A2 A3 …

Figure 1: Mapping From cwebp To mtpaint Data Structures

Figure 1 presents the mapping from the cwebp to the mtpaint data structures. CCC infers this mapping as matched arithmetic sequences. The first is $\langle \text{mem\_clip.img}[0] : 3k, \text{picture.argb} : 4k + 2\rangle$, which specifies that every fourth element of cwebp picture.argb starting at offset 2 maps to every third element of mtpaint mem_clip.img[0]. Two similar matched sequences complete the RGB mapping. The matched arithmetic sequence $\langle \text{mem\_clip.img}[1] : k, \text{picture.argb} : 4k + 3\rangle$ specifies the mapping for the alpha channel. Because these data structures are dynamically allocated arrays, CCC computes these sequences by walking the donor data structures to find corresponding values in the recipient data structures.

Global Variable Lifting: In general (and in our example), donor function may access global variables. To facilitate the insertion of the donor into the recipient name space, CCC replaces each global variable in the donor function with a pointer passed as a parameter into the transferred code. CCC generates appropriate values for these pointers at the insertion point before the transferred code is invoked.

```c
int mem_sel_rot(int dir, image_info *mem_clip__cc_,
  int* prog_stop__cc_, GtkWidget **progress_window__cc_,
  GtkWidget **progress_bar__cc_,
  undo_data **undo_freelist__cc_)
```

Listing 3: Lifting Globals

Listing 3 presents the updated `mem_sel_rot` interface in our example. All globals lifted by CCC are appended with the `__cc_` suffix. In Listing 3 `prog_stop__cc_`, `progress_window__cc_`, `progress_bar__cc_`, and `undo_freelist__cc_` are lifted globals accessed by GUI code `progress_init` and `progress_update` invoked by `mem_rotate`.

Irrelevant Functionality Removal: In some cases the transferred code may implement irrelevant functionality. The developer can enable CCC to remove irrelevant functionality by identifying parameters that relate to this functionality. To help the developer identify these parameters, CCC presents parameters with no translation to the name space of the recipient. In our example (Listing 3), the developer identifies `undo_freelist__cc_`, `progress_window__cc_`, `progress_bar__cc_`, and `prog_stop__cc_` as irrelevant parameters. These are parameters that were included as part of the `mem_rotate` GUI update code (Listing 2).

Starting with the developer identification of irrelevant parameters, CCC uses an interprocedural static analysis to identify and remove all transferred code that accesses irrelevant values (values derived from the irrelevant parameters). In our example, the analysis effectively excises the GUI code that was invoked by mtpaint. The irrelevant parameters are also removed from the `mem_sel_rot` interface.

Code Transfer: We next present the code (the input adapter, the call to the transferred code, and the output adapter) that CCC generates at the insertion point in the recipient. The input adapter starts by declaring and initializing the parameters to the transferred version of `mem_sel_rot` (Listing 4). These declarations include declarations of the original parameters and lifted global variables. These variables are declared as local variables in the recipient and passed into the transferred code as parameters. Here `mem_clip__cc_c` is a lifted global variable and `__cc_dir` is the original parameter to `mem_sel_rot`.

```c
int __cc_dir = 0;
image_info * __cc_mem_clip;
image_info __cc_mem_clip_init = {0};
__cc_mem_clip = &__cc_mem_clip_init;
unsigned int k;
```

Listing 4: Generated Parameter Declarations

Input Adapter: The transferred version of `mem_sel_rot` accesses data stored in the donor data representation. The input adapter allocates and populates the data structures that store the data in this representation. Specifically, the input adapter allocates and populates the `mem_clip.img[0]` BGR array that `mem_sel_rot` will access. The generated code uses the inferred matched arithmetic sequences to copy in the BGR data in loops that iterate over the allocated data structure (Listing 5).

Allocated data structures can include dynamically allocated arrays whose size depends on the input. In the generated code above, CCC derives a symbolic representation of the size of the data allocated in the donor as a function of the constants and input bytes that determine the size. It then translates this symbolic representation into the recipient name space to obtain the size of the allocated data. In this case the sizes are a function of the `picture.width` and `picture.height` variables in the recipient (Listing 5).

```
__cc_mem_clip->img[0] = malloc((picture.width * picture.height) * 3);
for (k = 0; 3*k < (picture.width*picture.height)*3; ++k)
     ((unsigned char *)__cc_mem_clip->img[0])[3*k+(0)] = ((unsigned char *)picture.argb)[4*k+(2)];
for (k = 0; 3*k < (picture.width*picture.height)*3; ++k)
     ((unsigned char *)__cc_mem_clip->img[0])[3*k+(1)] = ((unsigned char *)picture.argb)[4*k+(1)];
for (k = 0; 3*k < (picture.width*picture.height)*3; ++k)
     ((unsigned char *)__cc_mem_clip->img[0])[3*k+(2)] = ((unsigned char *)picture.argb)[4*k+(0)];
```

Listing 5: Allocating and Populating BGR Data Structure

The next step is to allocate and populate the alpha channel data structure. Once again, CCC uses an inferred arithmetic sequence to generate code that populates the allocated data structure (Listing 6).

```
__cc_mem_clip->img[1] = malloc(picture.width*picture.height);
for (k = 0; 1*k < picture.width * picture.height; ++k)
   ((unsigned char *)__cc_mem_clip->img[1])[1*k+(0)] = ((unsigned char *)picture.argb)[4*k+(3)];
```

Listing 6: Allocating and Populating the alpha Channel

The generated code next sets the remaining scalar parameters and invokes `mem_sel_rot` (Listing 7). `mem_clip.width` and `mem_clip.height` are derived from input bytes and translate to `picture.width` and `picture.height` in the recipient. `dir` and `mem_clip.bpp`, on the other hand, do not depend on input bytes. CCC therefore assigns these variables to the values (1 and 3) observed in the instrumented execution. In some transfers these values are the appropriate values and are untouched by the developer. In others the developer may wish to adjust the values to properly reconfigure the transferred functionality.

```
__cc_dir = 1;
__cc_mem_clip->bpp = 3;
__cc_mem_clip->width = picture.width;
__cc_mem_clip->height = picture.height;
mem_sel_rot(__cc_dir, __cc_mem_clip);
```

Listing 7: Scalar Parameters and Call to `mem_sel_rot`

Output Adapter: The output adapter copies the `mem_sel_rot` outputs into the corresponding data structures in the recipient (Listing 8). Once the copy is complete, the generated code deallocates any allocated donor data structures:

```
for (k = 0; 3*k < (picture.width*picture.height)*3; ++k)
     ((unsigned char *)picture.argb)[4*k+(2)] = ((unsigned char *)__cc_mem_clip->img[0])[3*k+(0)];
for (k = 0; 3*k < (picture.width*picture.height)*3; ++k)
     ((unsigned char *)picture.argb)[4*k+(1)] = ((unsigned char *)__cc_mem_clip->img[0])[3*k+(1)];
for (k = 0; 3*k < (picture.width*picture.height)*3; ++k)
     ((unsigned char *)picture.argb)[4*k+(0)] = ((unsigned char *)__cc_mem_clip->img[0])[3*k+(2)];
free(__cc_mem_clip->img[0]);
for (k = 0; 1*k < picture.width*picture.height; ++k)
     ((unsigned char *)picture.argb)[4*k+(3)] = ((unsigned char *)__cc_mem_clip->img[1])[1*k+(0)];
free(__cc_mem_clip->img[1]);
picture.width = __cc_mem_clip->width;
picture.argb_stride = __cc_mem_clip->width;
picture.height = __cc_mem_clip->height;
```

Listing 8: Copying Outputs Into Recipient Data Structures

In our example all of the outputs are stored in InOut variables, so the generated code uses the inferred mapping to copy the outputs back out from the donor data structures into the corresponding recipient data structures.

Other transfers (see Section 4) may write their outputs into (unmapped) Out parameters. In such cases CCC generates code that allocates the Out parameters and passes them into the transferred code, which writes its outputs into these parameters. The developer may then insert code that appropriately integrates the outputs into the recipient computation or find bridge functions in the donor and recipient that enable CCC to map and translate Out state (Section 3.2).

3 DESIGN AND IMPLEMENTATION

We next present the CCC design and implementation.

3.1 Instrumentation

We implement the CCC instrumentation using Valgrind [13]. The instrumentation augments the execution to record, for each Valgrind register, temp, or memory address, a symbolic expression for the value stored in the register, temp, or memory address. The symbolic expression records the complete computation of the value from constants and input file bytes. The instrumentation also records, for each pointer to a dynamically allocated block of memory, the symbolic expression for the size of the allocated memory block.

$$
\begin{aligned}
s \quad := \quad & \ell : x = \texttt{irrel} \mid \ell : x = c \mid \ell : p = \&y.f \mid \ell : x = y \oplus z \mid \\
& \ell : x = {*p} \mid \ell : {*p} = x \mid \ell : p = \texttt{malloc}(x) \mid \ell : x = \texttt{read}() \mid \\
& \ell : s' ;\, s'' \mid \ell : \texttt{if}\ (x)\ s'\ s'' \mid \ell : \texttt{while}\ (x)\ \{\ s'\ \}
\end{aligned}
$$

$$
s, s', s'' \in \texttt{Statement} \qquad f \in \texttt{Field}
$$

$$
x, y, z, p \in \texttt{Var} \qquad c \in \texttt{Int} \qquad \ell \in \texttt{Label}
$$

Figure 2: The Core Programming Language

We use the core language in Figure 2 to present the instrumentation algorithm. Following the Valgrind IR (as well as standard lowered program representations), the language is based on a load-/store model in which all computation takes place on named local variables $x, y, z, p \in \texttt{Var}$ (here local variables correspond to Valgrind registers and temps). Note that $\oplus$ represents an arbitrary binary operator. We also use the core language to present the irrelevant functionality removal algorithm (Section 3.3). The labels ℓ and statement $\ell : x = \texttt{irrel}$ are used only in the static analysis in Section 3.3.

$$
e, e', e'' \in \texttt{Exp} \qquad e := c \mid \text{byte}(c) \mid \text{cons}(\oplus, e', e'')
$$

$$
\mu : \texttt{Var} \cup \texttt{Addr} \to \texttt{Exp} \qquad \psi : \texttt{Addr} \to \texttt{Exp}
$$

$\langle \mu, \psi \rangle$	$\ell : x = c$	$\langle \mu[x \mapsto c], \psi \rangle$
$\langle \mu, \psi \rangle$	$\ell : p = \&y.f$	$\langle \mu, \psi \rangle$
$\langle \mu, \psi \rangle$	$\ell : x = y \oplus z$	$\langle \mu[x \mapsto \text{cons}(\oplus, \mu[y], \mu[z])], \psi \rangle$
$\langle \mu, \psi \rangle$	$\ell : x = {*p}$	$\langle \mu[x \mapsto \mu[\text{val}(p)]], \psi \rangle$
$\langle \mu, \psi \rangle$	$\ell : {*p} = x$	$\langle \mu[\text{val}(p) \mapsto \mu[x]], \psi \rangle$
$\langle \mu, \psi \rangle$	$\ell : p = \texttt{malloc}(x)$	$\langle \mu, \psi[\text{val}(p) \mapsto \mu[x]] \rangle$
$\langle \mu, \psi \rangle$	$\ell : x = \texttt{read}()$	$\langle \mu[x \mapsto \text{byte}(\text{fp})], \psi \rangle$

Figure 3: The CCC Instrumentation

Figure 3 presents the instrumentation algorithm. The instrumentation works with symbolic expressions $e, e', e'' \in \texttt{Exp}$. Each leaf is either a constant c or byte(c), the input file byte at offset c. The symbolic expression constructor cons($\oplus, e', e''$) creates a symbolic expression that represents $e' \oplus e''$.

The instrumentation maintains maps μ, which stores the symbolic expression for each variable and memory address, and ψ, which stores the sizes of allocated memory blocks. The notation $\langle \mu, \psi \rangle s \langle \mu', \psi' \rangle$ specifies how the instrumentation updates the two maps when s executes. Consider, for example, a store statement of the form $\ell : {*p} = x$. The instrumentation for this statement, $\langle \mu, \psi \rangle \ell : {*p} = x \langle \mu[\mathrm{val}(p) \mapsto \mu[x]], \psi \rangle$, updates μ to map the address in p (val(p)) to the symbolic expression for the value in x ($\mu[x]$) and leaves ψ unchanged. Note that read() reads and returns a single byte from the input file; `fp` denotes the input file offset of the last read byte.

When donor execution reaches the donor function or recipient execution reaches the insertion point, the instrumentation uses the debugging information in the executable and the two maps μ and ψ to generate log entries that identify the source-level expression, the value in the execution, and the symbolic expression for each piece of reachable state. During the execution of the donor function the instrumentation generates a log entry for each executed statement. Each entry identifies the computation that the statement performs as well as the source-level expressions, values, and symbolic expressions of all accessed state.

3.2 Input and Output Adapter Generation

Algorithm 1 presents the top-level CCC input adapter generation algorithm. Here $p_1, ..., p_n$ are the parameters passed to the transferred code. For each parameter p the algorithm emits the declaration for p, then invokes populate(p) to allocate and populate the donor data structure rooted at p.

```
inputAdapter($p_1, ..., p_n$)
  foreach $p \in \{p1, ..., p_n\}$ do
    emit type($p$) $p$ = {0};
    populate($p$)
```

Algorithm 1: Input Adapter Generation Algorithm

Data Structure Allocation and Population: Algorithm 2 presents the CCC populate(v) algorithm for allocating and populating the donor data structure rooted at v. Here v is a source-level expression, in the name space of the donor, that identifies the root of the tree to allocate and populate. If v is a primitive C data type (`int`, `char`, ...), a C structure, or a statically allocated array, the algorithm assumes that v has already been declared. The algorithm therefore emits only the code that populates v, either by translating v's symbolic expression into the recipient name space (if v is a primitive C data type) or by populating its elements (if v is a structure or statically allocated array). If v is a pointer to a structure, the algorithm emits code that declares the structure, sets v to point to the declared structure, and populates the fields of the structure.

Finally, if v is a pointer to an array, the algorithm first translates the allocated size of the array into the recipient name space. If v is an Out array, the algorithm emits code that allocates an array of the appropriate size and assigns the resulting pointer to v. The transferred code will then write the results into the newly allocated array.

If v is an In or InOut array, the algorithm finds matched arithmetic sequences m that capture the mapping from the recipient arrays to v. Here m is a set of tuples of the form $\langle v, s, o, a, t, p \rangle$, where s is the step and o is the offset for the donor array v and t and p are the step

```
populate($v$)
  if (isPrimitive($v$) emit $v$ = translate(expr($v$))
  if (isStruct($v$)) foreach $f \in$ fields($v$) do populate($v.f$)
  if (isArray($v$)) foreach $i \in [0..$ size($v$)$-1]$ do populate($v$[$i$])
  if (isPointerToStruct($v$))
    fresh $u$
    emit type(*$v$) $u$ = {0}; $v$ = &$u$;
    foreach $f$ in fields(*$v$) do populate($v$->$f$)
  if (isPointerToArray($v$))
    $l$ = translate(size($v$))
    if (isOut($v$)) emit $v$ = (type($v$)) malloc($l$);
    else
      $m$ = match($v$)
      if ($m = \emptyset$) abort
      else if ($m = \{\langle v, 1, 0, a, 1, 0 \rangle\}$) emit $v = a$;
      else
        emit { unsigned int k = 0;
        emit $v$ = (type($v$)) malloc($l$);
        foreach $\langle v, s, o, a, t, p \rangle \in m$ do
          emit for (k = $o$; k*$s$+$o$ < $l$ ; k+=$s$ )
          emit   $v$[k*$s$+$o$] = $a$[k*$t$+$p$];
        emit }
```

Algorithm 2: Allocating and Populating Donor Data Structures

and offset for the recipient array a (so that $v[k * s + o] = a[k * t + p]$ whenever $0 \leq k * s + o < l$, where l is the number of elements in v). If there is no match (i.e., $m = \emptyset$), the algorithm aborts and the transfer fails. If there is a direct match (i.e., $m = \{\langle v, 1, 0, a, 1, 0 \rangle\}$ for some array a in the name space of the donor), the algorithm simply sets $v = a$. Otherwise the algorithm translates the allocated size of v into the name space of the recipient, emits code that allocates a new array of the appropriate size and assigns the resulting pointer to v, and emits the for loops that iterate over the recipient arrays to populate the newly allocated donor array.

Constructs and Functions: The algorithm uses the following constructs and functions. Unless otherwise specified, v is a source-level expression in the name space of the donor and e is an application-independent expression over constants and input bytes.

- expr(v): the symbolic expression for the value of v. This expression is a function of program constants and input bytes, is generated by the instrumented execution of the donor, and is available in the log file from the donor execution.
- size(v): the number of elements in the array v. If v is a statically allocated array, size(v) is the (constant) declared size. If v is a dynamically allocated array, size(v) is the symbolic expression for the value passed to the call to `malloc` that allocated the array. The CCC instrumentation maintains this size information for all pointers to dynamically allocated memory blocks.
- type(v): the C type of the expression v, available via the debugging information in the donor executable.
- fields(v): the set of fields f in a structure v, available via the debugging information in the donor executable.

- translate(e): the CodePhage algorithm ([18], Figure 7) that translates the symbolic expression e (over constants and input bytes) into the name space of the recipient.
- eval(e): the concrete value of the symbolic expression e for the seed input file from the instrumented donor and recipient executions.
- value(v): the value of v in the donor or recipient execution, available in the log file from the instrumented donor or recipient execution. Here v is a source-level expression in either the donor or recipient name space.
- **fresh** u: a new, unused C variable name u that can be used to declare a new C structure with name u.
- **emit** s: emits a string s into the input adapter with variables and constructs from the algorithm appropriately substituted. So, for example, if p = `__cc_dir` and type(p) = `int`, **emit** type(p) p `= {0};` emits the string `int __cc_dir = {0};` into the input adapter.

```
Variables
  $a_1, ..., a_n$: arrays available in recipient name space
match($v$)
  $l$ = eval(size($v$))
  foreach $k \in [0..l-1]$ do matched[$k$] = false
  $m = \emptyset$
  for ($o = 0; o < l; o++$) do
    if (not matched[$s$])
      if (find min $\langle s,p,t \rangle$ over $1 \le i \le n$ such that
          value($v[k*s+o]$) = value($a_i[k*t+p]$)
        whenever $0 \le k*s+o < l$)
        matched[$k*s+o$] = true for $0 \le k*s+o < l$
        $m = m \cup \{\langle v,s,o,a_i,t,p \rangle\}$
      else return $\emptyset$
  return $m$
```

Algorithm 3: Donor and Recipient Array Matching

Array Matching Algorithm: Algorithm 3 presents the CCC algorithm that computes the data representation mapping between donor and recipient arrays. Given a donor array v and recipient arrays $a_1, ..., a_n$, it computes a set of matched arithmetic sequences $\langle v,s,o,a,t,p \rangle$ that translate the recipient arrays into the donor array. The algorithm traverses the unmatched elements of v. At each unmatched element (at offset o in v), it finds a tuple $\langle s,p,t \rangle$ and array a_i that implement a matched arithmetic sequence of corresponding equal elements of v and a_i. These equal elements occur at step s and offset o within v and step t and offset p within a_i so that $v[k*s+o] = a_i[k*t+p]$ whenever $0 \le k*s+o < l$.

At each unmatched offset o, the algorithm searches for the densest mapping into v by minimizing $\langle s,p,t \rangle$ over all of the possible matching arrays a_i. Here the minimum is taken with respect to the lexicographic ordering over $\langle s,p,t \rangle$ (so that $\langle s_1,p_1,t_1 \rangle \le \langle s_2,p_2,t_2 \rangle$ when $s_1 \le s_2$, $s_1 = s_2$ and $p_1 \le p_2$, or $s_1 = s_2, p_1 = p_2$, and $t_1 \le t_2$). The algorithm accumulates all of the matched arithmetic sequences into m and returns m if it matches all of the elements of v (and $\emptyset$ if it is unable to find a successful mapping).

Note that, unlike the data structure mapping for scalars, the presented algorithm finds matching array elements by comparing the concrete values that occur in the actual instrumented executions. CCC also implements a version that uses the symbolic expressions.

Output Adapter Generation: The CCC output adapter generation algorithm emits code that is essentially the inverse of the input adapter code — it traverses the donor data structures and applies the inverse of the input adapter data structure mapping to copy the values of InOut variables, fields, and arrays back into the corresponding recipient data structures. It also deallocates any In or InOut arrays that the input adapter allocated.

The output adapter leaves all Out state intact. It is the resposibility of the developer to write code to integrate the values stored in Out state into the recipient computation and to deallocate any dynamically allocated Out state.

Pointers Into the Middle of Allocated Data: As presented, the algorithms work with pointers that always point to the start of allocated data. We have implemented an extension for pointers that point into the middle of allocated data. The key is to change the range of array elements to match (as determined by l from Algorithm 3, line 4). Instead of using the allocated array size, CCC uses the log file information to find the range of elements accessed via the pointer during the execution of the donor function. The modified Algorithm 3 then checks that all elements in the accessed range match.

Out State Translations: We have also implemented an extension that uses pairs of *bridge functions*, one from each application, to translate Out state from the donor into the recipient data representation. Both bridge functions create internal data structures (these are Out state) that store common data derived from the seed input file. CCC, working as a general data representation translation algorithm, generates an output adapter that translates from the donor into the recipient data representation. The developer can then direct CCC to generate code that applies this output adapter to the Out data structures of the transferred code. This extension can be particularly useful when the primary data representation translation challenge involves Out state.

3.3 Irrelevant Functionality Removal

Language: Figure 2 (Section 3.1) presents the core language that we use to present our static analysis for removing irrelevant functionality. The language models a standard lowered intermediate program representation in which 1) nested expressions are lowered into sequences of statements of the form $\ell : x = y \oplus z$ (where x, y, and z are non-aliased local variables or temporaries and $\oplus$ is an arbitrary binary operator) and 2) all memory accesses are lowered into loads ($\ell : x = *p$) and stores ($\ell : *p = x$). Each statement contains a unique label $\ell \in$ `Label`.

One deviation from standard program representations is the statement $\ell : x =$ `irrel`, which sets the variable x to an irrelevant value. Such statements identify variables that contain irrelevant values. At the start of the analysis all irrelevant donor function parameters (as identified by the developer) are set to `irrel`.

Goal and Annotations: The goal of our analysis is to find all statements that may manipulate irrelevant values. The analysis works with a transfer function $F(s, \langle \sigma, I \rangle) = \langle \sigma', I' \rangle$. F takes a statement s, an abstract environment σ, and a set of labels I, which correspond to the set of statements that may manipulate irrelevant values. F produces a pair $\langle \sigma', I' \rangle$, where σ' and I' are the new abstract environment and new set of labels after executing s.

Statement s	**Transfer Function** $F(s, \langle\sigma, I\rangle)$		
$\ell : x$=`irrel`	$\langle\sigma[x \mapsto \{\texttt{irrel}\}]$	$, I \cup \{\ell\}\rangle$	
$\ell : x$=c, $\ell : x$=`read()`	$\langle\sigma[x \mapsto \{\texttt{rel}\}$	$, I\rangle$	
$\ell : p$=`malloc`(x)	$\langle\sigma[p \mapsto \{\texttt{rel}\}]$	$, I\rangle$	if $\texttt{irrel} \notin \sigma(x)$
	$\langle\sigma[p \mapsto \{\texttt{rel}, \texttt{irrel}\}]$	$, I \cup \{\ell\}\rangle$	otherwise
$\ell : x$=$y \oplus z$	$\langle\sigma[x \mapsto \{\texttt{rel}\}]$	$, I\rangle$	if $\texttt{irrel} \notin \sigma(y) \cup \sigma(z)$
	$\langle\sigma[x \mapsto \{\texttt{rel}, \texttt{irrel}\}]$	$, I \cup \{\ell\}\rangle$	otherwise
$\ell : p$=&$y.f$	$\langle\sigma[p \mapsto \left(\cup_{\langle\ell,\texttt{entry}\rangle\in\sigma(y)}\{\langle\ell, f\rangle\}\right)]$	$, I\rangle$	if $\texttt{irrel} \notin \sigma(y)$
	$\langle\sigma[p \mapsto \left(\cup_{\langle\ell,\texttt{entry}\rangle\in\sigma(y)}\{\langle\ell, f\rangle\}\right) \cup \{\texttt{irrel}\}]$	$, I \cup \{\ell\}\rangle$	otherwise
$\ell : x$=$*p$	$\langle\sigma[x \mapsto \cup_{\langle\ell,f\rangle\in\sigma(p)}\sigma(\langle\ell, f\rangle)]$	$, I\rangle$	if $\texttt{irrel} \notin$ $((\cup_{\langle\ell,f\rangle\in\sigma(p)}\sigma(\langle\ell, f\rangle)) \cup \sigma(p))$
	$\langle\sigma[x \mapsto (\cup_{\langle\ell,f\rangle\in\sigma(p)}\sigma(\langle\ell, f\rangle)) \cup \{\texttt{irrel}\}]$	$, I \cup \{\ell\}\rangle$	otherwise
$\ell : *p$=x	$\langle\sigma[\langle\ell, f\rangle \mapsto \sigma(x)]$	$, I\rangle$	if $\texttt{Loc} \cap \sigma(p) = \{\langle\ell, f\rangle\}$ and $\texttt{irrel} \notin \sigma(p) \cup \sigma(x)$
	$\langle\sigma[\langle\ell, f\rangle \mapsto \sigma(x)]$	$, I \cup \{\ell\}\rangle$	else if $\texttt{Loc} \cap \sigma(p) = \{\langle\ell, f\rangle\}$ and $\texttt{irrel} \in \sigma(p) \cup \sigma(x)$
	$\langle\sigma[\langle\ell_i, f_i\rangle \mapsto (\sigma(\langle\ell_i, f_i\rangle) \cup \sigma(x))]_{\langle\ell_i,f_i\rangle\in\sigma(p)}$	$, I\rangle$	else if $\texttt{irrel} \notin \sigma(p) \cup \sigma(x)$
	$\langle\sigma[\langle\ell_i, f_i\rangle \mapsto (\sigma(\langle\ell_i, f_i\rangle) \cup \sigma(x))]_{\langle\ell_i,f_i\rangle\in\sigma(p)}$	$, I \cup \{\ell\}\rangle$	otherwise
$s' ; s''$	$F(s'', F(s', \langle\sigma, I\rangle))$		
$\ell :$ `if` $(x)\ s'\ s''$	$\langle\text{Merge}(F_1(s', \langle\sigma, I\rangle), F_1(s'', \langle\sigma, I\rangle))$	$, F_2(s', \langle\sigma, I\rangle) \cup F_2(s'', \langle\sigma, I\rangle)\rangle$	if $\texttt{irrel} \notin \sigma(x)$
	$\langle\text{Merge}(F_1(s', \langle\sigma, I\rangle), F_1(s'', \langle\sigma, I\rangle))$	$, F_2(s', \langle\sigma, I\rangle) \cup F_2(s'', \langle\sigma, I\rangle) \cup \{\ell\}\rangle$	if $\texttt{irrel} \in \sigma(x)$
$\ell :$ `while` $(x)\{s'\}$	$\langle\text{Merge}(\sigma_{\texttt{fix}}, \sigma)$	$, I_{\texttt{fix}} \cup I\rangle$	if $\texttt{irrel} \notin \sigma(x)$
	$\langle\text{Merge}(\sigma_{\texttt{fix}}, \sigma)$	$, I_{\texttt{fix}} \cup I \cup \{\ell\}\rangle$	otherwise
	where $\langle\sigma_{\texttt{fix}}, I_{\texttt{fix}}\rangle = F(s', \langle\text{Merge}(\sigma_{\texttt{fix}}, \sigma), I_{\texttt{fix}} \cup I\rangle)$		

$$\text{Merge}(\sigma, \sigma') = \sigma[x \mapsto \sigma(x) \cup \sigma'(x)]_{x\in\texttt{Var}}[\langle\ell, f\rangle \mapsto \sigma(\langle\ell, f\rangle) \cup \sigma'(\langle\ell, f\rangle)]_{\langle\ell,f\rangle\in\texttt{Loc}}$$

Figure 4: Transfer Function for Irrelevant Code Removal. F_1 **and** F_2 **are the first and second canonical projection functions of** F**.**

F	$\in$	`TransFunc`	$=$	`Statement` $\times$ `AbstEnv` $\times$ `IrrelStmts` $\rightarrow$ `AbstEnv` $\times$ `IrrelStmts`
I	$\in$	`IrrelStmts`	$=$	$\mathcal{P}$(`Label`)
σ	$\in$	`AbstEnv`	$=$	(`Var` $\cup$ `Loc`) $\rightarrow$ `AbstValue`
v	$\in$	`AbstValue`	$=$	$\mathcal{P}$(`Loc` $\cup$ {`rel`, `irrel`})
$\langle\ell, f\rangle$	$\in$	`Loc`	$=$	`Label` $\times$ `Field`

Figure 5: Analysis Annotations

```
1  $F(\ell\colon y = \texttt{call}\ \mathit{func}\ x_1, \ldots, x_k, \langle\sigma, I\rangle)$
2   where definition($\mathit{func}$) = $\mathit{func}(a_1, a_2, \ldots, a_k)$ { $s$; return $b$ }
3     $\sigma' = \emptyset$
4     foreach $\langle\ell, f\rangle \in \texttt{Loc}$ do $\sigma' = \sigma'[\langle\ell, f\rangle \mapsto \sigma(\langle\ell, f\rangle)]$
5     foreach $i \in \{1, 2, \ldots, k\}$ do $\sigma' = \sigma'[a_i \mapsto \sigma(x_i)]$
6     $\langle\sigma'', I'\rangle = F(s, \langle\sigma', I\rangle)$
7     $\sigma''' = \sigma$
8     foreach $\langle\ell, f\rangle \in \texttt{Loc}$ do $\sigma''' = \sigma'''[\langle\ell, f\rangle \mapsto \sigma''(\langle\ell, f\rangle)]$
9     $\sigma''' = \sigma'''[y \mapsto \sigma''(b)]$
10    return $\langle\sigma''', I'\rangle$
```

Algorithm 4: Function Call Analysis Algorithm.

An abstract environment σ maps each of the variables and memory locations to an abstract value. An abstract value v is a set that contains abstract memory locations `rel` and/or `irrel`. Intuitively, v denotes the set of possible values that a variable or a memory location may take during the execution. The value may correspond to an address of a memory location, a relevant value (`rel`), or an irrelevant value (`irrel`).

Our analysis is field sensitive. Each abstract memory location is a pair $\langle\ell, f\rangle$, where ℓ is the label of the malloc statement that allocates the memory block that contains the location and f is the field of the memory location inside the allocated memory block. Each memory block contains a special field `entry` $\in$ `Field` which corresponds to the memory location in a memory block with no explicit field access.

Given a program s, CCC computes $F(s, \langle\sigma_0, \emptyset\rangle) = \langle\sigma, I\rangle$, where σ_0 is an initial environment that maps each of the variables and memory locations to an empty set. CCC then prunes away all statements whose labels are inside I. Note that if the label of an if statement or a while statement is inside I, CCC will recursively prune away all sub-statements of the statement as well.

Transfer Function: Figures 4 and 5 present the transfer function F and analysis annotations. The first column presents the statement s, the second column presents $\langle\sigma', I'\rangle = F(s, \langle\sigma, I\rangle)$. Here $\sigma[x \mapsto v]$ denotes the abstract environment obtained via changing the mapping of x to v in σ.

Interprocedural Analysis: Algorithm 4 presents the pseudo code for computing the transfer function F for a call statement $\ell\colon y = \texttt{call}\ \mathit{func}\ x_1, \ldots, x_k$, where $x_1, \ldots, x_k$ are the supplied parameters for the call and *func* is the called function with formal parameters $a_1, \ldots, a_k$. The body statement of *func* is s and the return statement in *func* returns the local variable b.

At lines 4-5 in Figure 4, the analysis transforms the initial abstract environment σ into the name space of called function *func*, and stores the transformed environment as σ'. Specifically, the analysis maps each formal argument a_i to the abstract value of the supplied argument x_i at line 5.

The analysis then performs the intraprocedural analysis on the body statement s with the transformed environment σ' and obtains a result pair $\langle\sigma'', I'\rangle$ at line 6. The analysis computes the final abstract environment σ''' by incorporating the effect of the memory accesses and the return value of *func* at lines 7-9. The analysis finally returns $\langle\sigma''', I'\rangle$ as the result.

Functionality Removal: Given the set of irrelevant statements I' produced by the interprocedural analysis, CCC applies a removal pass that deletes all statements s with labels $\ell \in I'$.

3.4 Code Extractor

The CCC code extractor identifies all code required to compile and execute the donor function and emits a single source code file that contains all of this code. This file can then be compiled and linked into the recipient. To preserve the source-level structure and promote readability, CCC transfers all code verbatim (i.e., it does not preprocess the transferred code).

Starting with the identified function to transfer, the code extractor (transitively) traces out compile- and run-time dependences to build a compile-time dependence graph. The nodes in this graph are code elements (type declarations, function declarations, and potentially invoked functions). The edges model compile-time dependences — there is an edge between two code elements if the compiler must process the first code element before the second for the second to compile successfully. The extractor topologically sorts this graph and, with the exception of system code elements from standard include files or system libraries, emits the code elements in the topological sort order. Instead of emitting code elements from standard include files, it emits code that includes the include file. It expects code from system libraries to be linked into the final executable.

To facilitate the mapping of program state between the donor and the recipient, the code extractor lifts all accessed global variables into the signature of the transferred code. The extractor walks the call graph to transitively identify and lift global variable accesses and update function declarations, definitions, and call sites.

4 EXPERIMENTAL RESULTS

We evaluate CCC on eight transfers between six applications: VLC 2.0.8 [6], MPlayer svn34540 [3], cwebp 0.3.1 [2], bmp2tiff 4.0.3 [1], ViewNior-1.4 [5] and mtPaint 3.40 [4]. We were familiar with VLC, cwebp, bmp2tiff (as part our prior work on automatic integer overflow discovery (DIODE [16]) and horizontal code transfer of security checks (CodePhage [18]). These applications work with standard image and video input files but provide distinct functionality. This fact, in combination with our previous success automatically transferring security checks [18], inspired the goal of automatically transferring useful functionality across applications that process the same input file. We identified MPlayer as a useful source of donor functionality as part of the research presented in this paper.

Figure 6 summarizes the results. There is a row in the table for each transfer. The first two columns identify the recipient and donor. The next column describes the transferred functionality, while the fourth column (Successful?) identifies if the transfer was successful. Here we count a transfer as successful if our manual analysis of the generated transfer indicates that the transfer will produce correct results for all input files (modulo any required developer integration code). The fifth column (CCC Time) specifies the wall clock time spent by CCC in automatically 1) finding a mapping between the donor and recipient data structures, 2) extracting the transferred code into a C source file, and 3) generating the adapters and inserting the donor code into the recipient.

The sixth column (Developer Code) specifies the code that the developer writes as part of the transfer. The In subcolumn identifies the number of unmapped In parameters (in Section 2, `mem_clip.bpp` and `dir`). CCC sets such parameters to the values observed during the instrumented execution, so there is no need to involve the developer to obtain working code, but in some cases the developer may wish to reconfigure the transferred functionality by changing these parameters. The Out subcolumn identifies the number of unmapped Out parameters. To incorporate the result into the computation, the developer assigns a recipient variable to each unmapped Out parameter (all such assignments comprise a single line of code). For all of the transfers except cwebp into mtpaint, which requires some additional scaffolding code (see Section 4), these lines of code are the only lines of code that the developer needs to write.

The seventh column (Transferred Lines of Code) presents the total lines of code transferred from the donor into the recipient, both the core lines of code (the lines of code in the top-level donor function) and the total lines of code — these include invoked code and declarations required for the transferred code to compile.

MPlayer → VLC: MPlayer and VLC are popular open-source media players written in C. These applications can post-process videos with filters so as to alter the video's (usually aesthetic) appearance.

We next discuss the rotate, mirror, EQ and Hue video filter functionality transfers from MPlayer into the Filter module in VLC. First, the developer identifies the donor function (`rotate`, `mirror`, `process_C` in the `vf_eq.c` file, or `process_C` in the `vf_hue.c` file). Second, the developer identifies the insertion point in the `Filter` function in VLC.

The CCC transfer times vary between 16 minutes, 28 seconds and 3 minutes, 3 seconds (Figure 6). All transfers involve two unmapped In parameters and a few (automatically generated) lines of code in addition to the core functionality. For the rotate transfer, the additional line of code is an include of `stddef.h`; the two unmapped In parameters are `bpp` (the bytes per pixel) and `dir` (the direction to rotate the frame). For the mirror transfer, the additional code comprises a `#define` and an include of `stddef.h`; the two unmapped In parameters are `bpp` and `fmt` (the pixel format of the video frame). For the EQ transfer, the additional line of code is an include of `stddef.h`; the unmapped In parameters are `brightness` and `contrast`, which control the brightness and contrast of the pixel transform. For the hue transfer, the additional code includes three files; the unmapped In parameters are `hue` and `sat`, which control the color shift and intensity of the pixel transform.

In each case, the developer writes code to assign Out parameters from the donor to appropriate recipient variables. For example, in the `vf_hue` transfer, the developer writes code that assigns `dst->p[1].p_pixels` and `dst->p[2].p_pixels` to the out array parameters `udst` and `vdst`. For all transfers, the transferred functionality does not require any lifted globals and there is a direct match between corresponding data structures (RGB to RGB). Both MPlayer and VLC build data structures with pointers into the middle of allocated blocks. The transfers therefore require the extension (Section 3.2) that enables CCC to generate such transfers.

mtPaint rotate image → cwebp: We describe this transfer in Section 2. The majority of the additional transferred code is code (transitively) invoked by the top-level rotate image function.

Recipient	Donor	Functionality	Successful?	CCC Time	Developer Code In	Developer Code Out	Transferred Lines of Code Core	Transferred Lines of Code Total
VLC	MPlayer	Rotate video filter	✓	6m12s	2	3	32	33
VLC	MPlayer	Horizontal mirror video filter	✓	6m20s	2	3	51	55
VLC	MPlayer	Horizontal EQ video filter	✓	16m28s	2	1	20	21
VLC	MPlayer	Hue/Saturation video filter	✓	3m37	2	2	24	27
cwebp	mtPaint	Rotate image	✓	4m24s	2	0	21	147
bmp2tiff	mtPaint	Rotate image	—	—	—	—	—	—
ViewNior	mtPaint	Horizontal Flip	✓	23m9s	0	0	38	39
mtpaint	cwebp	WebP format Reader	✓	4m35s	4	2	38	9965

Figure 6: Summary of CCC experimental results

mtPaint rotate image → bmp2tiff: We next discuss CCC's inability to transfer the `mem_flip_h` function from mtPaint into bmp2tiff. The bmp2tiff data structures store the image pixels in reverse row-major order — the last row of pixels appears first in the array, followed by the next row, and so on, with the first row of pixels appearing at the end of the array. mtPaint, on the other hand, stores the pixels in row-major order — the first row of pixels appears first in the array, then the second row, with the last row of pixels appearing last in the array. CCC does not support the data structure translation required to implement this transfer.

mtPaint horizontal flip → ViewNior: We next discuss the horizontal flip functionality transfer from mtPaint (an open source painting program) to the ViewNior image viewer. The developer first identifies the `mem_flip_h` function in mtpaint, which takes an image buffer and flips it horizontally. The developer then identifies an insertion point in the ViewNior `gdk_pixbuf_animation_new_from_file` function after the generic image loader.

In addition to the core functionality, CCC transfers an extra include of `stddef.h`. The transferred functionality does not require any lifted globals and there is a direct match between the recipient and donor data representations. There is a single unmapped In parameter (`bpp`).

cwebp → mtpaint: cwebp is the Google conversion program for the WebP image format. mtpaint is a popular image manipulation application. Mtpaint supports several formats (PNG, GIF, etc.) but not WebP.

The developer first identifies a bridge function from each application, specifically `read_png` from webp and `load_png` from mtpaint. Both functions read a PNG file and decode the file into the internal data representation of their respective application (used for all file formats). CCC executes cwebp and mtpaint on the same PNG file to create an output adapter that translates the cwebp internal representation (a BGR and alpha channel-separated format) into the mtpaint internal representation (an ARGB format). The translation is the inverse of the translation presented in Section 2).

The developer next identifies the `ReadWebP` function in cwebp, which reads in a WebP image and decodes it into the cwebp internal data representation. The developer also adds the necessary scaffolding to support loading a new format in mtpaint. Specifically, the developer adds an enum type for the WebP format, registers the WebP format with the file format handlers, and adds the magic number checking code for detecting image formats. Finally, the developer adds a `load_webp` function to handle the WebP image loading and creates an insertion point inside `load_webp`. In summary, the developer added 44 lines as scaffolding for the new format. It took the authors, who were familiar with the mtpaint code, 1 hour to add this code.

The developer finally identifies the cwebp `ReadWebP` function as the transferred code and directs CCC to apply the output adapter from the bridge functions as part of the transfer.

CCC successfully completes the cwebp transfer in 4 minutes and 9 seconds. The core functionality consists of 38 lines of code. The total transfer consists of 9965 lines of code. This transfer includes all of the transitively invoked code that reads in the WebP file and decodes it into the cwebp internal data structures. This is the largest transfer — implementing a new file format involves significantly more code than the image processing functionality from the other transfers. The transferred functionality does not require any lifted globals.

Public Availability: Source code for all of the transfers evaluated in this paper is publicly available at https://people.csail.mit.edu/rinard/paper/fse17.codecarboncopy.zip.

5 THREATS TO VALIDITY

We worked with all but one of our benchmark applications in previous research projects [16, 18]. All of these applications read standard image or video input files but provide different functionality. These characteristics inspired the goal of functionality transfer with a data structure mapping driven by data derived from a common input file and stored in application data structures. Our focus on arrays (as opposed to other data structures) was motivated by the fact that these applications store much of their relevant data in arrays. We anticipate that other applications may present different code transfer challenges that inspire the development of new code transfer techniques.

Our current data structure mapping algorithm supports linear gather/scatter relationships across multiple arrays. It does not support other data structures such as hash tables, lists, trees, or developer-defined recursive data structures more generally. It is an open question whether the approach will generalize to handle functionality transfers that involve data structures other than arrays (or even more complex array mappings).

CCC requires both applications to 1) read the same input file, 2) store the relevant input data in accessible application data structures, and 3) execute both the donor function and the recipient insertion point. We anticipate that many desirable functionality transfers may involve applications that work with different inputs. Supporting such transfers will require new techniques that can operate without shared data to drive the mapping. CCC also derives the mapping from a single execution. If other executions (either on different inputs or nondeterministic reexecutions on the same input) store data in different data structures or data formats, the transferred code may access the wrong data and produce a wrong result.

CCC uses a static analysis to identify and transfer all code that the transferred code may execute on any execution, including code not executed by the input file used to drive the data transfer. While this strategy promotes transfer of the full functionality, including functionality not exercised by the input file used to drive the data mapping, it is also possible that unexecuted but transferred code may not execute properly in the new recipient context.

Our approach requires the developer to 1) identify the transferred functionality in the source code of the donor, 2) identify the insertion point in the source code of the recipient, and 3) identify irrelevant data to drive the irrelevant functionality removal. The approach therefore requires at least some familiarity with both the donor and recipient source code bases.

Our current instrumentation does not attempt to bound the size of the derived symbolic expressions. Long-running applications may therefore build up very large symbolic expressions over time. Our current implementation mitigates this issue by turning off donor instrumentation at the end of the donor function and recipient instrumentation when execution reaches the insertion point. If either of these points occurs deep in the execution, large symbolic expression sizes may limit scalability.

6 RELATED WORK

We discuss related work in code transfer and program repair.

μScalpel: μScalpel [8] uses test-driven genetic programming to adapt code for transfer from a donor to a recipient program. Like CCC, μScalpel requires the developer to identify the code to transfer and the code insertion point. μScalpel uses genetic programming to search for a variable mapping from the name space of the donor to the parameters of the inserted code that enables the inserted code to produce correct outputs for test input files. It also uses genetic programming to automatically find and prune undesirable or irrelevant functionality (CCC, in contrast, uses a static analysis). μScalpel therefore requires a full test suite with correct input/output pairs that specify the desired behavior of the augmented recipient.

CCC, in contrast, uses a single instrumented execution to directly compute a mapping between the recipient and donor data structures. It therefore requires a single input, no outputs, and does not perform multiple validation runs. This usage scenario reduces developer overhead and worked well for the transfers evaluated in this paper. But (as expected for a dynamic technique that does not explore all executions on all inputs) it has the potential to produce transfers that may not work for executions or inputs not used to drive the transfer.

Unlike CCC, μScalpel does not support data representation translations. Several of our transfers require nontrivial data representation translations and thus lie inherently outside the reach of μScalpel.

CodePhage: CodePhage [18] implements a fully automated technique that finds and transfers checks between applications to eliminate security vulnerabilities. CodePhage is driven by an input that exposes the vulnerability and leverages that input to automatically identify the transferred check and the insertion point in the recipient. CCC incorporates a more sophisticated data structure translation algorithm that can automatically infer *matched arithmetic sequences* that translate data stored in dynamically allocated arrays whose size depends on the input. CCC can therefore successfully transfer complete computations over dynamically allocated arrays (as opposed to checks as in CodePhage).

Program Repair: Several program repair projects leverage information available within the same or other programs to repair defects in a given program [7, 9–12, 14, 15, 17, 19]. Prophet learns characteristics of correct code from human patches, obtaining these patches from the revision history repositories of multiple applications [12]. It uses these patches to learn a universal probabilistic model of patch correctness, which it then applies to identify and prioritize correct patches that repair defects in new programs. History driven program repair ranks candidate patches based on how well they match past correct patches [9]. Genesis learns patch transforms from previous successful patches, then uses the inferred transforms to generate new patches [10, 11].

FixMeUp automatically computes an interprocedural access-control template (ACT) and uses the ACT to find and correct faulty access-control logic [19]. FixMeUp then presents the transformed program to the developer, who decides whether to accept the proposed correction. FixMeUp works only within a single program, not across multiple programs.

Researchers have developed a technique that is provided with input validation and sanitization PHP functions and uses these functions to obtain new PHP validation and sanitization functions [7]. The technique is based on semantic analysis and synthesis of string operations. Its scope is therefore limited to string validation and sanitization functions with input/output relationships that it can accurately analyze and represent with finite state automata.

ClearView is an automatic patch generation system that observes normal executions to learn invariants that characterize safe behavior [14]. It deploys monitors that detect crashes, illegal control transfers and out of bounds write defects. In response, it selects a nearby invariant that the input that triggered the defect violates, and generates patches that take a repair action to enforce the invariant.

CCC differs from program repair that it does not aspire to correct defects — it instead transfers new functionality from a donor application into a recipient application to enhance the functionality that the recipient application implements.

7 CONCLUSION

Code copying and sharing between applications is a productive software development strategy that is currently hindered by the need for manual reworking to adapt the donor code into its new context in the recipient. To support this process, CCC implements automatic data representation and naming translation between recipient and donor and a static analysis that automatically identifies and removes code that is irrelevant in the recipient. This functionality enables CCC to eliminate much of the manual rework otherwise required to successfully transfer code at the source level between applications.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments on earlier drafts of the paper. This research was supported by DARPA (Grant FA8750-14-2-0242).

REFERENCES

[1] The bmp2tiff project home page. http://www.libtiff.org/.
[2] Cwebp. https://developers.google.com/speed/webp/docs/cwebp.
[3] The mplayer project home page. http://www.mplayerhq.hu/.
[4] mtpaint. http://mtpaint.sourceforge.net/.
[5] Viewnoir - the elegant image viewer. http://xsisqox.github.io/Viewnior/.
[6] VLC media player. http://www.videolan.org/.
[7] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 225–236, 2014.
[8] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 257–269. ACM, 2015.
[9] X. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 213–224, 2016.
[10] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms and search spaces for automatic patch generation systems. Technical Report MIT-CSAIL-TR-2016-010, MIT, July 2016.
[11] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.
[12] F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful human patches. 2016.
[13] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. PLDI '07, 2007.
[14] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. SOSP '09. ACM, 2009.
[15] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS*, pages 37–48, 2009.
[16] S. Sidiroglou, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *ASPLOS*, 2015.
[17] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the general track, 2005 USENIX annual technical conference: April 10-15, 2005, Anaheim, CA, USA*, pages 149–161. USENIX, 2005.
[18] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. 2015.
[19] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.