Published in “Conference on Compiler Construction” (CC), 2005, pp. 172–187.

Data Slicing:
Separating the Heap into Independent Regions

Jeremy Condit and George C. Necula

Department of Electrical Engineering and Computer Science
University of California, Berkeley
{jcondit,necula}@cs.berkeley.edu

Abstract. In this paper, we present a formal description of *data slicing*, which is a type-directed program transformation technique that separates a program’s heap into several independent regions. Pointers within each region mirror the structure of pointers in the original heap; however, each field whose type is a base type (e.g., the integer type) appears in only one of these regions. In addition, we discuss several applications of data slicing. First, data slicing can be used to add extra fields to existing data structures without compromising backward compatibility; the CCured project uses data slicing to preserve library compatibility in instrumented programs at a reasonable performance cost. Data slicing can also be used to improve locality by separating “hot” and “cold” fields in an array of data structures, and it can be used to protect sensitive data by separating “public” and “private” fields. Finally, data slicing can serve as a refactoring tool, allowing the programmer to split data structures while automatically updating the code that manipulates them.

1 Introduction

When maintaining a large software project, a seemingly trivial change to a data structure can be largely intractable due to the amount of code that depends upon that data structure’s layout. When programmers wish to modify a data structure, they must weigh the benefits of these modifications against the time required to modify the program and the risk of introducing new bugs. Compilers face a similar challenge; for example, a compiler may wish to alter one of the data structures in the program it is compiling without violating data layout assumptions made by precompiled code. Such changes require a principled approach that can achieve the desired goal automatically and without changing the program’s semantics.

This paper introduces *data slicing*, a program transformation technique that addresses this problem. Given an existing program, data slicing produces a new program that computes the same result while splitting its data structures among several memory regions. This transformation allows the programmer or the compiler to factor out portions of a data structure that must reside in a different place in memory.

Data slicing can be used to preserve backward compatibility after a program transformation. For example, a transformation that adds new fields to existing data structures may make the program incompatible with precompiled libraries; data slicing can be used to separate these new fields from the old ones, allowing the original program's data structures to retain their original layout. As the prototypical example of this application, we show how to instantiate data slicing in the context of the CCured project [4] to enable extensive run-time checking of C programs while maintaining compatibility with precompiled libraries.

Data slicing can also be applied to performance optimizations. For example, data slicing can be used to produce an improved implementation of the instance interleaving optimization [13], which interleaves the fields of several objects in order to place frequently-accessed fields in the same cache line.

Finally, data slicing can be applied to security problems. For example, data slicing can move function pointers to a separate memory region to make it more difficult for an attacker to overwrite them. In general, data slicing acts as a refactoring tool that simplifies the task of making global changes to data structures.

This paper offers two main contributions. First, it presents a formal description of the type-directed data slicing transformation on a simple imperative language. Second, it discusses several applications of this technique, including CCured, instance interleaving, and security-related transformations.

2 Data Slicing

Data slicing is a program transformation that separates a program's heap into independent regions. The input to this transformation is a program whose base types (e.g., integer types) have been annotated with region names. The goal of data slicing is to produce a new program that computes the same result as the original program while splitting the data structures in the program's heap into independent regions. Each region must contain only the data that has been annotated for that region, as well as any pointers that are necessary for keeping track of that data. We focus on the case when the regions must be *independent*, in the sense that there are no pointers that cross region boundaries; however, we will show how to relax this requirement in Sections 2.5 and 3.1.

To achieve these goals, data slicing ensures that each region in the program mirrors the structure of the original program's heap. For example, if the original program's heap contains a linked list with data whose fields are annotated with multiple regions, then each region of the transformed program's heap will contain a linked list of the same length containing the list data for that region. More precisely, there is an injective mapping m_i from objects in region R_i to objects in the original program's heap at a given point in execution. An object A in region R_i contains the fields of object $m_i(A)$ whose type is labelled with region R_i. Furthermore, if object A points to object B in region R_i, then object $m_i(A)$ points to object $m_i(B)$ in the original program's heap. Note that base fields (i.e., fields whose type is a base type) will be stored in exactly one region according to the relevant type annotation, whereas each pointer field may be split into several pointer fields, one in each region where it is necessary. Essentially, data slicing *separates* base fields and *replicates* some pointer fields.

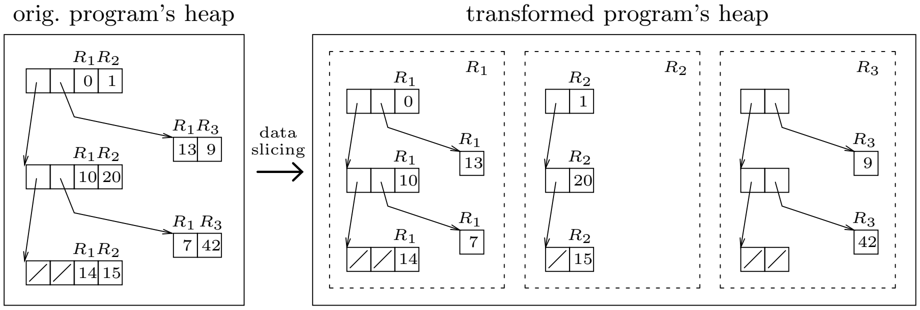

Fig. 1. Illustration of data slicing's effects. The dashed lines delimit the regions within the transformed program's heap

types	$\tau ::= \texttt{int}\ R_i \mid t \mid \tau\ \texttt{ptr} \mid \texttt{struct}\{\ \ldots f_j : \tau_j;\ \ldots\ \}$
l-expressions	$l\ ::= x \mid l.f \mid {*e}$
expressions	$e ::= n \mid \texttt{null} \mid \texttt{new}\ \tau \mid e_1\ \texttt{op}\ e_2\ \mid \&l \mid l$
	$\mid \texttt{cast}\langle \texttt{int}\ R_j \hookrightarrow \texttt{int}\ R_i \rangle\ e \mid \texttt{cast}\langle \tau\ \texttt{ptr}\ \hookrightarrow \texttt{int}\ R_i \rangle\ e$
commands	$c ::= l := e \mid f(x) \mid \texttt{let}\ x : \tau\ \texttt{in}\ c \mid c_1; c_2$
	$\mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c$
definitions	$d ::= f(x : \tau)\ \{c\} \mid \texttt{type}\ t = \tau$
programs	$p ::= d\ p \mid d$

Fig. 2. A C-like language used as the basis for discussing data slicing

Figure 1 illustrates the effects of data slicing. This figure shows a program's heap at a specific point in time before and after data slicing. The base fields in the original heap are annotated with region names R_1, R_2, and R_3. In the transformed heap, each region contains a data structure with the same shape as the original, except that it contains only the base fields for that region as well as any pointers needed to access those fields. Note that we have eliminated entire objects from region R_2; thus, the mapping m_2 is injective but not surjective.

The remainder of this section presents the data slicing transformation formally. First, we introduce a C-like language (Section 2.1), and then we define data slicing on types and programs (Sections 2.2 and 2.3). Then, we discuss first-class functions and partial data slicing (Sections 2.4 and 2.5).

2.1 Language

Figure 2 shows an imperative language that will be the basis for our discussion of data slicing. The types in this language include integer types, pointer types, structure types, and named types (t). We use `void` as a shorthand for `struct`$\{\}$. Base types (here, the integer type) have a region qualifier that indicates the region where this data should be placed. Region names are R_1 through R_n; throughout this paper, n will refer to the number of available regions.

The syntax for l-expressions (l) and expressions (e) is based loosely on that of C. Binary operations ("`op`" in the grammar) include arithmetic and comparison, and they may only be applied to integers in the same region. We permit casts between integers in different regions as well as casts from pointers to integers.

Commands (c) include standard imperative constructs. We will use $l_1, ..., l_k := e_1, ..., e_k$ as syntactic sugar for simultaneous assignment where all right-hand sides are evaluated before any assignments occur. Function calls have no explicit return value; instead, the programmer must pass a pointer to the result variable as an argument. At the top level, a program (p) is a list of definitions (d) of functions and types. Type definitions allow recursive types to be defined. To simplify the presentation, we defer the discussion of function pointers to Section 2.4. The complete static semantics for this language can be found in Appendix A.

Using this language, we can write down the types of the objects in Figure 1. In the original program's heap, there are two types of objects, t and t', as follows:

$$\begin{aligned}
&\texttt{type}\ t\ \ = \texttt{struct}\{\ p_1 : t\ \texttt{ptr};\ p_2 : t'\ \texttt{ptr};\ f_1 : \texttt{int}\ R_1;\ f_2 : \texttt{int}\ R_2;\ \}\\
&\texttt{type}\ t' = \texttt{struct}\{\ f_1 : \texttt{int}\ R_1;\ f_3 : \texttt{int}\ R_3;\ \}
\end{aligned}$$

To translate a C program to this language, the programmer must add region annotations to each field or variable whose type is a base type. The programmer must either eliminate unsafe pointer casts or limit the use of data slicing to the safe portions of the program using a technique discussed in Section 2.5.

2.2 Transformation of Types

We now define the data slicing transformation on types, which will guide the rest of the transformation. In the previous section, we defined two types, t and t', for the objects on the left-hand side of Figure 1. Data slicing splits each object of type t or t' into three objects, one in each region. The resulting types are:

$$\begin{aligned}
&\texttt{type}\ t_1 = \texttt{struct}\{\ p_1 : t_1\ \texttt{ptr};\ p_2 : t'_1\ \texttt{ptr};\ f_1 : \texttt{int}\ R_1;\ \}\\
&\texttt{type}\ t_2 = \texttt{struct}\{\ p_1 : t_2\ \texttt{ptr};\ f_2 : \texttt{int}\ R_2;\ \}\\
&\texttt{type}\ t_3 = \texttt{struct}\{\ p_1 : t_3\ \texttt{ptr};\ p_2 : t'_3\ \texttt{ptr};\ \}
\end{aligned}$$

$$\begin{aligned}
&\texttt{type}\ t'_1 = \texttt{struct}\{\ f_1 : \texttt{int}\ R_1;\ \}\\
&\texttt{type}\ t'_2 = \texttt{void}\\
&\texttt{type}\ t'_3 = \texttt{struct}\{\ f_3 : \texttt{int}\ R_3;\ \}
\end{aligned}$$

Note that the pointers of type t `ptr` in the original heap have been split into pointers of type t_1 `ptr`, t_2 `ptr`, and t_3 `ptr` in their respective regions. Similarly, the pointers of type t' `ptr` have been split into pointers of type t'_1 `ptr` and t'_3 `ptr` in regions R_1 and R_3. It is unnecessary to have a pointer of type t'_2 `ptr` in region R_2 because t' contains no data labelled with region R_2 (i.e., t'_2 is `void`). Note that integer fields only appear in the region to which they have been assigned.

Formally, we define a mapping TSlice_i from types in the original program to the corresponding types in region R_i of the transformed program (see Figure 3). The first rule in this definition says that the sliced type for region R_i contains

$$\mathsf{TSlice}_i(\texttt{int}\ R_j) = \begin{cases} \texttt{int}\ R_j & \text{if } i = j \\ \texttt{void} & \text{otherwise} \end{cases}$$

$$\mathsf{TSlice}_i(\tau\ \texttt{ptr}) = \begin{cases} \mathsf{TSlice}_i(\tau)\ \texttt{ptr} & \text{if } \mathsf{TSlice}_i(\tau) \neq \texttt{void} \\ \texttt{void} & \text{otherwise} \end{cases}$$

$$\mathsf{TSlice}_i(\texttt{struct}\{\ \ldots\ f_j : \tau_j;\ \ldots\ \}) = \texttt{struct}\{\ \ldots\ f_j : \mathsf{TSlice}_i(\tau_j);\ \ldots\ \}$$

$$\mathsf{TSlice}_i(t) = t_i$$

$$\mathsf{VSlice}(\tau) = \texttt{struct}\{\ \ldots\ r_i : \mathsf{TSlice}_i(\tau);\ \ldots\ \}$$

Fig. 3. Data slicing transformation for types. Note that we omit `void` fields from the resulting structure types

only those base types that are annotated with region R_i. The rules for structures and pointers recursively apply TSlice_i, and the rule for named types transforms a named type t into its corresponding named type t_i in region R_i. (In Section 2.3, we will define t_i to be $\mathsf{TSlice}_i(\tau)$, where τ is the original definition of t.)

Two optimizations occur during this transformation. First, when the result is a structure type, we omit fields of type `void`. Second, we omit pointers that cannot be used to reach any data in region R_i, as shown in the "otherwise" case for pointer types. For example, $\mathsf{TSlice}_1(\texttt{int}\ R_2\ \texttt{ptr ptr}) = \texttt{void}$ rather than `void ptr ptr`, since this type contains no information from region R_1. Of course, TSlice_2 yields $\texttt{int}\ R_2\ \texttt{ptr ptr}$, as desired.

TSlice_i gives the transformation for a specific region; however, at certain points in the program (variables and formal parameters), we must gather the sliced data into one structure containing the data from all regions. In Figure 3, VSlice gives the type of this merged structure. For example, a variable or formal parameter whose type is a pointer type would be transformed into a structure containing one pointer for each region where the pointer's sliced type is not `void`.

2.3 Transformation of Programs

In Figure 4, we show the transformation for the remaining syntactic constructs.

PSlice and DSlice transform programs and definitions. Formal parameters are transformed with VSlice, so they include data from all regions. For type definitions, we create one named type for each region.

CSlice defines data slicing for commands. Function calls are unchanged, since the argument variable's type will have been transformed with VSlice. For conditionals and loops, we transform the guard expression with ESlice_1, which slices an expression with type $\texttt{int}\ R_1$. (The guard expression must have this type.)

The assignment command is the key part of this transformation: essentially, the transformed program performs the corresponding assignment in each region where the type being assigned is not `void`. Since each region's assignment operation is performed separately, the rules for expressions and l-expressions are

$$
\begin{aligned}
\mathsf{PSlice}(d\ p) &= \mathsf{DSlice}(d)\ \mathsf{PSlice}(p)\\
\mathsf{PSlice}(d) &= \mathsf{DSlice}(d)\\[4pt]
\mathsf{DSlice}(f(x:\tau)\ \{c\}) &= f(x:\mathsf{VSlice}(\tau))\ \{\mathsf{CSlice}(c)\}\\
\mathsf{DSlice}(\texttt{type}\ t=\tau) &= \texttt{type}\ t_1=\mathsf{TSlice}_1(\tau)\ \ldots\ \texttt{type}\ t_n=\mathsf{TSlice}_n(\tau)\\[4pt]
\mathsf{CSlice}(f(x)) &= f(x)\\
\mathsf{CSlice}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) &= \texttt{if}\ \mathsf{ESlice}_1(e)\ \texttt{then}\ \mathsf{CSlice}(c_1)\ \texttt{else}\ \mathsf{CSlice}(c_2)\\
\mathsf{CSlice}(\texttt{while}\ e\ \texttt{do}\ c) &= \texttt{while}\ \mathsf{ESlice}_1(e)\ \texttt{do}\ \mathsf{CSlice}(c)\\
\mathsf{CSlice}(\texttt{let}\ x:\tau\ \texttt{in}\ c) &= \texttt{let}\ x:\mathsf{VSlice}(\tau)\ \texttt{in}\ \mathsf{CSlice}(c)\\
\mathsf{CSlice}(c_1;c_2) &= \mathsf{CSlice}(c_1);\mathsf{CSlice}(c_2)\\
\mathsf{CSlice}(l:=e) &= \mathsf{LSlice}_{i_1}(l),..,\mathsf{LSlice}_{i_k}(l):=\mathsf{ESlice}_{i_1}(e),..,\mathsf{ESlice}_{i_k}(e)\\
&\quad \text{where } \{i_1,..,i_k\}=\{i\in\{1,..,n\}\mid \mathsf{TSlice}_i(\mathsf{TypeOf}(e))\neq\texttt{void}\}\\[4pt]
\mathsf{ESlice}_i(n) &= n\\
\mathsf{ESlice}_i(\texttt{null}) &= \texttt{null}\\
\mathsf{ESlice}_i(e_1\ \texttt{op}\ e_2) &= \mathsf{ESlice}_i(e_1)\ \texttt{op}\ \mathsf{ESlice}_i(e_2)\\
\mathsf{ESlice}_i(\texttt{cast}\langle\texttt{int}\ R_j\hookrightarrow\texttt{int}\ R_i\rangle e) &= \texttt{cast}\langle\texttt{int}\ R_j\hookrightarrow\texttt{int}\ R_i\rangle\ \mathsf{ESlice}_j(e)\\
\mathsf{ESlice}_i(\texttt{cast}\langle\tau\ \texttt{ptr}\ \hookrightarrow\texttt{int}\ R_i\rangle e) &= \texttt{cast}\langle\mathsf{TSlice}_i(\tau\ \texttt{ptr})\hookrightarrow\texttt{int}\ R_i\rangle\ \mathsf{ESlice}_i(e)\\
\mathsf{ESlice}_i(\texttt{new}\ \tau) &= \texttt{new}\ \mathsf{TSlice}_i(\tau)\\
\mathsf{ESlice}_i(\&l) &= \&\mathsf{LSlice}_i(l)\\
\mathsf{ESlice}_i(l) &= \mathsf{LSlice}_i(l)\\[4pt]
\mathsf{LSlice}_i(x) &= x.r_i\\
\mathsf{LSlice}_i(l.f) &= \mathsf{LSlice}_i(l).f\\
\mathsf{LSlice}_i(*e) &= *\mathsf{ESlice}_i(e)
\end{aligned}
$$

Fig. 4. Data slicing transformation for programs, definitions, commands, expressions, and l-expressions, using n regions

defined with respect to a single region, and they assume that the sliced type of the expression in the given region is not `void`.

For example, suppose we want to transform the command $(*x).p_2 := y$, where x has type t `ptr` and y has type t' `ptr`. (We use the types t and t' defined in Section 2.1.) The rule for assignment yields $\mathsf{CSlice}((*x).p_2 := y) = \mathsf{LSlice}_1((*x).p_2), \mathsf{LSlice}_3((*x).p_2) := \mathsf{ESlice}_1(y), \mathsf{ESlice}_3(y)$. Thus, we will perform the corresponding assignment in regions R_1 and R_3, since $\mathsf{TSlice}_1(t'\ \texttt{ptr}) \neq \texttt{void}$ and $\mathsf{TSlice}_3(t'\ \texttt{ptr}) \neq \texttt{void}$, but not in region R_2, since $\mathsf{TSlice}_2(t'\ \texttt{ptr}) = \texttt{void}$.

Now consider ESlice_i and LSlice_i, the slicing operations for expressions and l-expressions, respectively. Here, we slice with respect to a specific region; for example, when transforming a variable reference, we select the component of that variable corresponding to the region in question. Continuing the example above, we have $\mathsf{LSlice}_1((*x).p_2) = (*x.r_1).p_2$ and $\mathsf{LSlice}_3((*x).p_2) = (*x.r_3).p_2$. Note that this slicing operation could not have been performed in region R_2, since $\mathsf{TSlice}_2(t)$ does not have a field called p_2. However, the assignment rule prevents us from calling ESlice_2 in this case, since $\mathsf{TSlice}_2(t'\ \texttt{ptr}) = \texttt{void}$. The final result for the example is $\mathsf{CSlice}((*x).p_2 := y) = (*x.r_1).p_2, (*x.r_3).p_2 := y.r_1, y.r_3$.

The integer cast expression computes a single integer value in region R_j using ESlice_j, casts this integer to region R_i, and completes the computation in region R_i. Since we only move a single integer value between regions, this operation preserves the invariant that there are no inter-region pointers.

Finally, note that the pointer cast expression targets a specific region, and this choice affects the result of the transformation. For example, the expression $\texttt{cast}\langle t\ \texttt{ptr}\ \hookrightarrow \texttt{int}\ R_1\rangle\ x$ would be transformed to $\texttt{cast}\langle t\ \texttt{ptr}\ \hookrightarrow \texttt{int}\ R_1\rangle\ x.r_1$, whereas $\texttt{cast}\langle t\ \texttt{ptr}\ \hookrightarrow \texttt{int}\ R_3\rangle\ x$ would become $\texttt{cast}\langle t\ \texttt{ptr}\ \hookrightarrow \texttt{int}\ R_3\rangle\ x.r_3$. After data slicing, these expressions will yield different integers; thus, when comparing two pointers, the programmer must ensure that they were obtained by casting to the same region. In this example, we cannot cast to region R_2 because our typing rules require that we cast to a region where $\mathsf{TSlice}_i(\tau\ \texttt{ptr}) \neq \texttt{void}$.

2.4 Handling First-Class Functions

Since data slicing splits data but not code, we cannot split a function among n different regions in the same way that we can split a pointer. Rather, function types are handled in the same manner as integer types: by adding a region qualifier. To implement this scheme, we add a function type, a function name expression, a function cast expression, and a new function invocation command.

$$\begin{aligned} \tau &::= \ldots \mid \tau\ \texttt{fn}\ R_i \\ e &::= \ldots \mid f \mid \texttt{cast}\langle \tau\ \texttt{fn}\ R_j \hookrightarrow \tau\ \texttt{fn}\ R_i\rangle\ e \\ c &::= \ldots \mid e(x) \end{aligned}$$

In the function type $\tau\ \texttt{fn}\ R_i$, the type τ refers to the type of the argument to the function. Next, we add new rules to our type and program transformations:

$$\begin{aligned} \mathsf{TSlice}_i(\tau\ \texttt{fn}\ R_j) &= \begin{cases} \mathsf{VSlice}(\tau)\ \texttt{fn}\ R_i & \text{if } i = j \\ \texttt{void} & \text{otherwise} \end{cases} \\ \mathsf{ESlice}_i(f) &= f \\ \mathsf{CSlice}(e(x)) &= \mathsf{ESlice}_1(e)(x) \end{aligned}$$

Function argument types are transformed with VSlice. Function names are unchanged, and function invocation retrieves the function from region R_1 as required by our type system. Function casts (not shown) resemble integer casts.

Unfortunately, this approach does not suffice for applications where data slicing is used to preserve backward compatibility. In these applications, we start with a program whose fields are all labelled with region R_1. When we add new fields, we label them with region R_2 so that data slicing will separate these fields from the original data structures, thus preserving the original layout of region R_1. However, the original layout of region R_1 may contain function types. We cannot label these types with region R_2, because data slicing would remove them from region R_1, breaking backward compatibility. However, we cannot keep them in region R_1, because the sliced type in R_1 may differ from the original type.

To solve this problem, we allow the programmer to introduce wrapper functions. These functions have the appropriate type for the original data layout, and they are responsible for calling the transformed function with arguments from all regions. In the above example, we can annotate the function type with region R_2, and then data slicing will place a wrapper function of the appropriate type in region R_1. The wrapper's implementation is application-specific.

For example, function types and expressions may be transformed as follows:

$$\mathsf{TSlice}_i(\tau\ \mathtt{fn}\ R_j) = \begin{cases} \mathsf{VSlice}(\tau)\ \mathtt{fn}\ R_i & \text{if } i = j \\ \mathsf{TSlice}_i(\tau)\ \mathtt{fn}\ R_i & \text{otherwise} \end{cases}$$

$$\mathsf{ESlice}_i(f) = \begin{cases} f & \text{if TypeOf}(f) = \tau\ \mathtt{fn}\ R_i \\ f_i & \text{otherwise} \end{cases}$$

The function f_i is a wrapper for function f in region R_i. This wrapper function takes an argument of type $\mathsf{TSlice}_i(\tau)$, which it uses to call f. Since this data corresponds to only one of the fields that make up $\mathsf{VSlice}(\tau)$, the wrapper function must fill in the rest of the fields in an application-specific manner. We will see an example of this approach in the CCured case study (Section 3.1).

2.5 Partial Data Slicing

The variant of data slicing presented so far splits the base fields of a data structure as well as all objects that directly or indirectly point to these base fields. In many cases, this additional slicing is wasteful; for example, when using data slicing to preserve library compatibility, we need not split objects that will not be shared with a library.

To solve this problem, we introduce an extension that allows pointer and structure types to be given region annotations. A pointer of type $\tau\ \mathtt{ptr}\ R_1$ would be split into as many as n pointers using the original rules, but all of these pointers would be stored in region R_1, despite the fact that they point to other regions. Because all components of this pointer appear in one region, types that contain this pointer do not necessarily need to be split. In a sense, we introduce a limited form of inter-region pointer in exchange for the ability to restrict data slicing to a small portion of the program. In fact, this extension can be used to derive CCured's technique for restricting its compatible metadata representation [4]. Due to space constraints, we omit the remaining details.

3 Case Studies

3.1 CCured

CCured [4, 10] is a program transformation system designed to guarantee memory safety in C programs through a combination of static analysis and run-time checks. To perform its run-time checks, CCured adds metadata to pointers, altering the layout of the program's data structures. Unfortunately, this new layout is incompatible with precompiled libraries, which proved to be a major obstacle when applying CCured to large software systems such as `bind` and `OpenSSH`.

To solve this problem, CCured can separate its metadata from the original program's data, placing this metadata in a parallel structure [4]. Data slicing generalizes this technique, as discussed in Section 4. In this section, we show how data slicing can be instantiated for CCured, demonstrating how one can solve data structure backward compatibility problems with data slicing.

CCured classifies pointers into one of several pointer kinds, which determine the metadata required by a given pointer. We will consider three CCured pointer kinds: `SAFE` pointers, which carry no metadata, `SEQ` ("sequence") pointers, which carry array bound information, and `RTTI` ("run-time type information") pointers, which carry an integer identifying the dynamic type of the pointer.

CCured infers these pointer kinds based on pointer usage, and then it implements them by transforming them into C structures, as follows:

$$\begin{aligned}
&\mathsf{Rep}(\mathtt{int}) &&= \mathtt{int}\ R_D \\
&\mathsf{Rep}(\tau\ \mathtt{ptr}\ \mathtt{SAFE}) &&= \mathtt{struct}\{\ p : \mathsf{Rep}(\tau)\ \mathtt{ptr};\ \} \\
&\mathsf{Rep}(\tau\ \mathtt{ptr}\ \mathtt{SEQ}) &&= \mathtt{struct}\{\ p : \mathsf{Rep}(\tau)\ \mathtt{ptr};\ b : \mathtt{int}\ R_M;\ e : \mathtt{int}\ R_M;\ \} \\
&\mathsf{Rep}(\tau\ \mathtt{ptr}\ \mathtt{RTTI}) &&= \mathtt{struct}\{\ p : \mathsf{Rep}(\tau)\ \mathtt{ptr};\ t : \mathtt{int}\ R_M;\ \} \\
&\mathsf{Rep}(\tau\ \mathtt{fn}) &&= \mathsf{Rep}(\tau)\ \mathtt{fn}\ R_M
\end{aligned}$$

Given a type annotated with CCured pointer kinds, the Rep function gives the representation of that type as a C type. For example, `SAFE` pointers are represented by a single pointer, whereas `SEQ` pointers also carry bounds information. Rep adds region qualifiers as appropriate: R_D for data, R_M for metadata.

To make these data structures compatible with existing libraries, we can apply the data slicing transformation after the CCured transformation. In previous work [4], we introduced functions called C and Meta to describe the types of the separated data and metadata structures; the interested reader can verify that $\mathsf{C} = \mathsf{TSlice}_D \circ \mathsf{Rep}$ and that $\mathsf{Meta} = \mathsf{TSlice}_M \circ \mathsf{Rep}$. (Note that in this paper, we use integer types instead of pointer types for the b and e fields.)

For function types, we use the wrapper function scheme from Section 2.4. Since Rep uses region R_M for all function types, the transformed functions (which take both data and metadata as arguments) are always stored in and retrieved from R_M. In region R_D, we place a wrapper whose type matches the original type of the function. This wrapper is responsible for looking up (or generating) appropriate metadata for its arguments before calling the transformed function.

Example of Data Slicing in CCured The C library functions `sendmsg` and `recvmsg` take as a parameter a pointer to a `msghdr`, which in turn contains an array of `iovecs`. A simplified declaration for these structures is as follows:

$$\begin{aligned}
\mathtt{type}\ \mathit{iovec} &= \mathtt{struct}\{\ \mathit{iov\_base} : \mathit{data}\ \mathtt{ptr}\ \mathtt{RTTI};\ \mathit{iov\_len} : \mathtt{int};\} \\
\mathtt{type}\ \mathit{msghdr} &= \mathtt{struct}\{\ \mathit{msg\_iov} : \mathit{iovec}\ \mathtt{ptr}\ \mathtt{SEQ}; \\
&\qquad\qquad\ \mathit{msg\_iovlen} : \mathtt{int};\ \mathit{msg\_flags} : \mathtt{int};\}
\end{aligned}$$

The type *data* is some unspecified type; for simplicity, we assume it contains no metadata. Figure 5 shows how data slicing separates the CCured metadata from this data structures. Once separated, the data portion can be passed directly to C library functions.

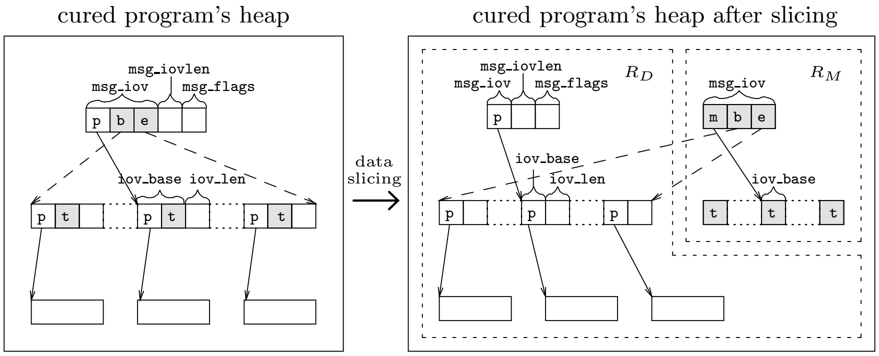

Fig. 5. Illustration of data slicing in CCured. CCured's metadata, shown in gray, is separated into the metadata region, R_M. The base and end fields are integers, not pointers, which is why they are allowed to "point" across region boundaries. These "pointers," which are drawn with dashed lines, are compared but never dereferenced

Performance CCured has been applied to several large systems programs (e.g., `OpenSSH`, `bind`, `ftpd`, `sendmail`, `Apache` modules) for which the ability to maintain compatibility with precompiled libraries was essential. In order to determine the impact of data slicing, we also applied CCured to simple benchmarks (`olden` [2] and `ptrdist` [1]) that can be cured without using data slicing.

These experiments were conducted on a 2.4 GHz Pentium 4 with 1 GB of memory running Linux 2.6.6. The results are reported in Table 1. The second and third columns show the average execution time (in seconds) of five runs of the cured program, with and without data slicing. Standard deviations were negligible in all cases. The third column shows the ratio of the sliced version to the unsliced version. The fourth column shows the percentage of pointers in the program text that required CCured metadata (i.e., were not `SAFE`). The final column indicates the percentage of pointers in the program text that were split into two pointers (i.e., one in each region). These percentages include all pointer types and all variables, since each variable's address is potentially a pointer.

The impact of data slicing on execution time was minimal for most of these benchmarks. The only three cases that had more than a 1% slowdown were `anagram`, `em3d`, and `mst`. The worst performance by far was shown by `em3d`, which had a 63% slowdown. For such cases, it is possible to restrict data slicing to only those portions of the program where it is necessary (see Section 2.5), thus minimizing the overall performance impact.

There is a rough correspondence between the number of pointers needing metadata and the number of pointers that need to be split into two pointers. Recall that a pointer will be split into two pointers if there is CCured metadata reachable from that pointer; thus, these two numbers will be correlated. While these static counts give a rough estimate of the performance impact of data slicing, they are not always reliable (compare `anagram` and `bh`); naturally, data slicing's performance depends significantly on how pointers are used at run time.

Table 1. Results for `ptrdist` and `olden`. We show execution time (in seconds) and the static percentage of pointers needing metadata and of pointers that were split. We omit `ptrdist`'s `bh` benchmark, since it uses CCured's `WILD` pointer, whose current implementation is not amenable to data slicing

Test	**Cured**	**Sliced**	**Ratio**	**Meta**	**Split**
anagram	3.001	3.329	1.10	12%	11%
ft	2.164	2.140	0.99	2%	1%
ks	2.617	2.597	0.99	12%	6%
yacr2	0.197	0.199	1.01	11%	12%
bh	3.592	3.572	0.99	20%	13%
bisort	1.906	1.915	1.00	3%	2%
em3d	0.275	0.449	1.63	6%	18%
health	1.305	1.303	1.00	3%	2%
mst	0.651	0.677	1.04	3%	14%
perim	2.106	2.106	1.00	0%	0%
power	3.584	3.583	1.00	2%	4%
treeadd	0.417	0.420	1.01	3%	3%
tsp	2.162	2.160	1.00	0%	0%

3.2 Instance Interleaving

Data slicing can also be used to implement compiler optimizations. To illustrate this application, we consider the instance interleaving optimization described by Truong et al. [13]. Instance interleaving is a data layout technique that clusters frequently-accessed ("hot") fields from a number of instances of a data structure, improving cache performance. Unfortunately, the original implementation required programmer intervention and had significant restrictions on the use of these structures. Data slicing provides an alternative implementation that addresses these problems.

Truong et al. presented instance interleaving using the following structure:

$$\texttt{type}\ t = \texttt{struct}\{\ a : \texttt{int};\ b : \texttt{int};\ c : \texttt{int};\ d : \texttt{int};\ \}$$

Assume that fields `a` and `c` are accessed far more frequently than fields `b` and `d`. To apply instance interleaving using the original approach, we would separate the "hot" fields and add padding (represented by an ellipsis):

$$\texttt{type}\ t = \texttt{struct}\{\ a : \texttt{int};\ c : \texttt{int};\ \ldots\ b : \texttt{int};\ d : \texttt{int};\ \}$$

Now, we allocate these objects from an array that is sized according to the amount of padding. "Hot" fields are stored in the first half of the array, and "cold" fields are stored in the second half. The padding represents the portions of the array that do not belong to this particular instance. The top half of Figure 6 shows how a pointer of type t `ptr` points to an instance of the structure t that is part of an interleaved array. The padding in the structure corresponds

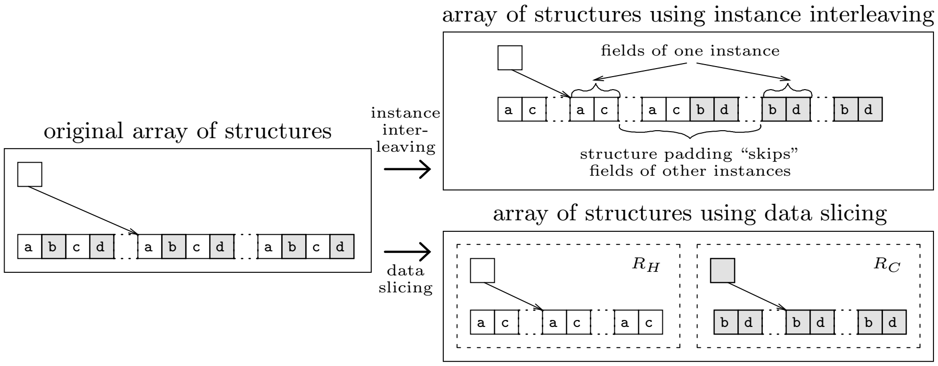
array of structures using instance interleaving
fields of one instance
a c a c a c b d b d b d
structure padding "skips"
fields of other instances
original array of structures
a b c d a b c d a b c d
instance
inter-
leaving
data
slicing
array of structures using data slicing
R_H
R_C
a c a c a c
b d b d b d

Fig. 6. Two implementations of instance interleaving. "Hot" fields are white, and "cold" fields are gray

to the fields of the other interleaved instances. The programmer allocates from this array by calling *ialloc*, a library function that manages the array.

This implementation requires that the programmer modify only the structure declaration and the allocation sites. However, pointer arithmetic, structure assignment, and static allocation are either prohibited or extremely wasteful.

Data slicing offers an alternative implementation that solves these problems. To use data slicing, we would assign "hot" and "cold" fields to different regions:

$$\texttt{type}\ t = \texttt{struct}\{\ a : \texttt{int}\ R_H;\ b : \texttt{int}\ R_C;\ c : \texttt{int}\ R_H;\ d : \texttt{int}\ R_C;\ \}$$

After data slicing, the "hot" and "cold" fields will appear in different regions. If we allocate objects from an array, the "hot" fields of several instances will be allocated adjacent to one another, as shown in the bottom half of Figure 6.

The data slicing approach has many advantages. First, there is minimal programmer intervention required, which eliminates an opportunity for introducing bugs. Second, it is possible to use pointer arithmetic, structure assignment, static allocation, and dynamically-sized arrays. Finally, we can have more than two classes of fields; for example, we could group fields that tend to be accessed at the same time and then assign one region to each group.

The primary disadvantage of the data slicing implementation is that in some cases, data slicing introduces an additional pointer. In Figure 6, the pointer into the array has been split into two pointers, one for the "hot" region and one for the "cold" region. This splitting arises because data slicing makes no assumptions about the size of the array. However, if we restrict data slicing using the technique from Section 2.5, then the pointers to the "hot" and "cold" parts of a data structure can be stored in the same region, without splitting the data structure that contains them. Truong et al. report speedups of 1.08 to 2.52 when using instance interleaving and reordering some fields; the overhead of the extra pointer required by data slicing should be comparatively small.

3.3 Security Applications

In this section, we present three examples that demonstrate how data slicing can be applied to security problems.

First, data slicing can be used to isolate function pointers in a program's heap. Function pointers can be a security vulnerability because an errant write that changes a function pointer could allow an attacker can gain control of the processor [15]. To solve this problem, it is not sufficient to add an extra level of indirection to function pointers: we could replace pointers of type τ `fn ptr` with pointers of type τ `fn ptr ptr`, but overwriting this new pointer is still a security vulnerability. Instead, we can label all function pointers with a special region name and apply data slicing. As a result, all function pointers and all pointers that indirectly point to them will be placed in this region, reducing the chances that an attacker can overwrite them.

Second, data slicing can protect sensitive data (such as a password) that is stored in virtual memory. Normally, the programmer must ensure that this data is not paged to disk; otherwise, an attacker who has access to the page file could recover the secret data [8]. If the user annotates sensitive data with a specific region name, then data slicing will separate this data into a region that can be marked as non-pageable. Here, data slicing automates a task that would otherwise be a tedious refactoring exercise.

Finally, suppose the programmer wishes to share portions of an application's data structures with an untrusted party. If the programmer labels public and private fields appropriately, data slicing will separate these fields into independent data structures. Since data slicing disallows inter-region pointers, the user is guaranteed that private data is not accessible from shared public data. Indeed, because the private data is stored in a completely separate memory region, it could be protected by the virtual memory system as well.

In general, data slicing provides the programmer with a refactoring tool. The programmer can label fields that need to be removed from a data structure, and data slicing will automatically make the desired change throughout the program.

4 Related Work

This work originated in the design of CCured's compatible metadata representation [4]. Unfortunately, the design of this compatible representation was largely ad-hoc and would be difficult to adapt for other purposes; in addition, the original presentation only showed how to transform types. Data slicing, as presented in this paper, provides a framework for applying this transformation in a much more general setting. In addition, we improve over previously published work by showing how to handle first-class functions, by allowing the transformation to split the heap into more than two regions, and by providing a detailed discussion of the program transformation itself. Finally, we show how this technique can be applied to other problems.

Structure splitting [3] separates infrequently-accessed fields by adding an extra level of indirection to a data structure. Data slicing provides an alternative approach, as shown in the instance interleaving example. Unfortunately, structure splitting is inappropriate for solving backward compatibility problems, since it adds an extra pointer to the original structure after removing fields.

Intensional polymorphism [5–7] is an approach to compiling polymorphism that allows type information to be used at run time. This technique allows a compiler to use efficient data representations while preserving type safety. Data slicing solves a similar problem, since it allows the compiler to refactor data structures automatically. Also, many of these approaches to intensional polymorphism represent types as terms in parallel with expressions; data slicing provides such parallel structures for arbitrary data.

One alternative approach to preserving backward compatibility is to use a global splay tree to store metadata [9]. However, this strategy was prohibitively expensive in CCured, since it altered the asymptotic complexity of some simple test cases. Data slicing allows constant-time metadata lookup in most cases; global lookups are only needed by wrapper functions at library boundaries. Another alternative is to factor runtime checks into a "shadow process" that executes on another processor [11]. Data slicing has several advantages over this approach: it is type-directed, handles first-class functions, requires less overhead, and requires only one processor.

Program slicing, which was introduced by Weiser [14] and later surveyed by Tip [12], extracts only those portions of a program that are relevant to computing the value of a particular variable at a particular program point. Data slicing does not discard any code; rather, it separates data in the heap into independent regions. However, there is some similarity: program slicing preserves statements that indirectly affect the value of the specified variable, and data slicing preserves pointers from which data in a given region is reachable.

5 Conclusions

In this paper, we have introduced data slicing, a program transformation that separates the heap into several independent regions. Using this technique, we can add new fields to a data structure without interfering with backward compatibility, and we can also implement compiler optimizations in a principled manner. In addition, we can implement security-related program transformations. Future work includes investigating ways to make data slicing work in the presence of unsafe pointer casts and automating the task of constructing wrapper functions for function pointers.

We believe that the data slicing technique is a promising approach to a number of common software engineering problems. It is particularly useful in combination with other automated program transformations, since it simplifies the task of improving these programs while preserving backward compatibility. As automated program transformations become more popular in practice, we believe that this technique will find a wide range of additional applications.

Acknowledgements

Thanks to Matt Harren, Scott McPeak, and Westley Weimer, whose work on CCured made this work possible. This research was supported in part by NSF Grants CCR-0085949, CCR-0326577, CCF-0524784, an NSF Graduate Research Fellowship, and an Alfred P. Sloan Foundation Research Fellowship. The information presented here does not necessarily reflect the position or the policy of the sponsors and no official endorsement should be inferred.

References

1. Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.
2. Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University Department of Computer Science, June 1996.
3. Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.
4. Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.
5. Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, pages 301–312, 1998.
6. Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, 1999.
7. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, 1995.
8. Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft, 2002.
9. Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *AADEBUG*, 1997.
10. George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002.
11. Harish Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.
12. Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
13. Dan N. Truong, François Bodin, and André Seznec. Improving cache behavior of dynamically allocated data structures. In *IEEE PACT*, pages 322+, 1998.
14. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.
15. Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *SIGSOFT International Symposium on Foundations of Software Engineering*, pages 307–316, 2003.

A Static Semantics

This section gives the static semantics for the language presented in this paper, including first-class functions. The environment Γ maps variables to types. A program is well-typed if the body of every function f type-checks with initial environment Γ_f, which maps f's argument to its type. FieldType(τ, f) gives the type of field f in the structure type τ. ArgType(f) gives the type of f's argument. The predicate HasComponent(τ, i) indicates whether there is a base type in region R_i that is reachable from τ, and it holds if and only if $\mathsf{TSlice}_i(\tau) \neq \texttt{void}$. This latter fact is required by the translation of the pointer-to-integer cast.

Expressions

$$\frac{}{\Gamma \vdash n : \texttt{int}\ R_i} \quad \frac{}{\Gamma \vdash \texttt{null} : \tau\ \texttt{ptr}} \quad \frac{}{\Gamma \vdash \texttt{new}\ \tau : \tau\ \texttt{ptr}}$$

$$\frac{\Gamma \vdash l : \tau}{\Gamma \vdash \&l : \tau\ \texttt{ptr}} \quad \frac{\tau = \mathrm{ArgType}(f)}{\Gamma \vdash f : \tau\ \texttt{fn}\ R_i} \quad \frac{\Gamma \vdash e_1 : \texttt{int}\ R_i \quad \Gamma \vdash e_2 : \texttt{int}\ R_i}{\Gamma \vdash e_1\ \texttt{op}\ e_2 : \texttt{int}\ R_i}$$

$$\frac{\Gamma \vdash e : \texttt{int}\ R_j}{\Gamma \vdash \texttt{cast}\langle \texttt{int}\ R_j \hookrightarrow \texttt{int}\ R_i \rangle\ e : \texttt{int}\ R_i} \quad \frac{\Gamma \vdash e : \tau\ \texttt{ptr} \quad \mathrm{HasComponent}(\tau, i)}{\Gamma \vdash \texttt{cast}\langle \tau\ \texttt{ptr} \hookrightarrow \texttt{int}\ R_i \rangle\ e : \texttt{int}\ R_i}$$

L-Expressions

$$\frac{x \in \mathrm{Dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma \vdash l : \tau_1 \quad \tau_2 = \mathrm{FieldType}(\tau_1, f)}{\Gamma \vdash l.f : \tau_2} \quad \frac{\Gamma \vdash e : \tau\ \texttt{ptr}}{\Gamma \vdash *e : \tau}$$

Commands

$$\frac{\Gamma \vdash l : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash l := e} \quad \frac{\Gamma \vdash e : \tau\ \texttt{fn}\ R_1 \quad \tau = \Gamma(x)}{\Gamma \vdash e(x)} \quad \frac{\Gamma \vdash e : \texttt{int}\ R_1 \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

$$\frac{\Gamma \vdash e : \texttt{int}\ R_1 \quad \Gamma \vdash c}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c} \quad \frac{\Gamma[x \mapsto \tau] \vdash c}{\Gamma \vdash \texttt{let}\ x : \tau\ \texttt{in}\ c} \quad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2}$$