Patching software remains a key defensive technique for mitigating flaws and vulnerabilities. We posit that studying the mechanics of fixes for vulnerabilities (i.e., security patches) can lead to insight about common principles and patterns for fixing software flaws. Such insights are of particular value to efforts to automatically deploy patches — especially at runtime to live software (e.g., for the purposes of hot patching or self-healing). Our study of these patch semantics provides some insight into the limitations of hot patching (in other words, what kinds of code and data constructs are too complicated to safely hot patch). Knowing the limitations of hot patching can help avoid blind or uncertain deployment of such fixes and potentially aid a system administrator in triaging which patches to deploy (either in a traditional restart–oriented fix or in a hot patch).

One of the biggest challenges of applying patches on the run is the introduction of statements modifying data structures. The challenge arises when trying to dynamically patch these statements in the source code layer of the application — since already existing data structures reside in the object code layer of the application. In this abstract we refer to the object code layer as the process address space of an application and one of our goals is to make it an easy environment for updating data structures.

The process address space consists of a collection of bytes that represents a running process, with little organization or means to distinguish its different elements. The largely unstructured nature of the process address space frustrates the analysis of its content for frameworks (e.g., debugging, vulnerability analysis, dynamic analysis) aiming to dynamically modify elements of a running process. As illustrated in Figure 1, we can use already existing mechanisms (e.g., pmap(1), proc/map) to view the structure of the process address space, this view is primarily of ABI systems and of limited use for other analysis. Our intent is to introduce a different organization that facilitates the task of hot–patching existing structures in the process address space. We cannot do this, however, without an exhaustive analysis of the statements, patches introduce, that create conflicts between the process address space and the source code layer hot–patching frameworks update. More precisely, the statements that update data structures. We call these statements data operations.

There are two main goals in our investigation, both related to the data operations (or data semantics) of security patches. Our first goal is to create an automated procedure for classifying patches as feasible to hot–patch (i.e., hot–patchable) or not. Our second goal is to develop a system for organizing and hot–patching the process address space of running applications using the patches our procedure classifies as hot–patchable. We present the DPL system, a system that can dynamically update data structures in a running process after organizing its data structures at runtime. The novelty of the DPL system resides in using a database for organizing the data structures inside a process address space and database queries (i.e., a data patch object) for updating them.

The DPL system instruments the process address space of an application by following a procedure to recognize, organize, update, and export (ROUE) the application’s data structures. The novelty of the DPL system resides in transforming different types of data structures into data that we are able to store inside a database, and thus we are able to update by using common database queries (i.e., a data patch object). In other words, DPL organizes live in–memory data structures in different tables inside a database and uses the metadata (e.g., size, type, name, value, address) of these data structures to populate the tables. We are then able to use database operations in the form of database queries to update the data structures and eventually export them back to the application. The system does all of this dynamically (i.e., at runtime) without stopping or interrupting the running process. Our research consists
of a manual process for creating a database query from a
security patch and an automated process for applying the
database query to the database that represents the running
application. We then export all the updated data structures from
the database to the running application without interrupting it.

Before developing the DPL system, however, we create
an automated procedure using machine learning techniques
that analyzes the common elements and implications of our
dataset of patches (we explicitly exclude from our study the
consideration of “general” patches e.g., feature addition). The
main purpose of the modeling and analysis is to help determine
whether a patch contains elements that are likely to cause
instability or incorrect operation if the patch is applied to the
running system. We use this analysis as the ground truth for
building the system to organize and update the data structures
inside the process address space of running applications.

In order to automate this procedure, we manually study
over 140 patches to get a data corpus that works as an input
to four machine learning algorithms: neural networks, naïve
Bayes classifiers, support vector machines, and decision trees.
We then use subsets of the data corpus as training and testing
datasets and we automate a procedure for classifying patches
as hot–patchable or not hot–patchable. If a patch is classified
as hot–patchable then it means that we can hot–patch its
data structures using the ROUE process of our system. We
plan to deploy the system in the future as a test framework
where patch–developers can test if their security patches can
be hot–patched or not. In Figure 2, we present the a heat map
representation for a subset of our dataset that includes a label
that tells if the patch (i.e., each row in the matrix) is hot–
patchable or not. At the end, we analyze this dataset with our
machine learning algorithms and decide the best technique to
automate our classification.

We then evaluate the ROUE process by organizing and
updating the data structures in the process address spaces of six
test cases in a test suite we define. We use a set of common data
operation patterns, in their database query form, we gathered
from the analysis of our dataset. DPL is capable of updating
programs fast with little CPU overhead and it is also capable
of updating several types of data structures including primitive
and user–defined types (e.g. a linked list).

For future work, we want to dynamically apply (i.e., hot–
patch) the security patches we find data–patchable to their
counter piece software. As of now, we are able to organize
the data structures in the process address space of ten core
utilities found in Linux distributions and update them using
database queries. We present how the results of the organiza-
tion of these utilities in a database in Table I. However, we
are not yet able to dynamically apply security patches, that
our automated procedure classifies as hot–patchable, to more
complex applications (e.g., Firefox, Samba, Apache HTTPD).

Fig. 2. A heat map representation of our data corpus. Each cell represents the
value of a feature, the darker the cell is the bigger the value of the feature is.
The last feature represents our manual classification as a feasible (i.e., data–
patchable) or not patch. If the cell for this feature is black then the patch is
data–patchable, if not then it is not.