Building a Forensic Computing Language

Vassil Roussev
Department of Computer Science
University of New Orleans
New Orleans, LA, 70148, USA
vassil@roussev.net

Abstract—The primary goal of this discussion is to motivate the need for the development of a domain-specific language (DSL) focused on the requirements of forensic and security analysis.

We argue that, at present, there is no adequate mechanism that a) allows analysts to specify the forensic computation as a tool-agnostic, logical sequence of steps; b) provides a formal specification for tool developers; and c) seamlessly integrates different available tools to provide a complete and extensible solution.

We present an initial design sketch for a forensic DSL called nugget, and use it to illustrate the ideas behind our approach.

I. INTRODUCTION

Over the past decade, digital forensic techniques have become an established part of the cyber security cycle—from data recovery, to incident response, to deep analysis of persistent threats. Yet, the actual process of (digital) forensic analysis lacks any sort of description that is both formal and useful. This is problematic from just about any point of view we can conceive of:

• Legal. In legal proceedings, by definition, the processing and analysis must be transparent and third-party verifiable. With an annual compound growth of 26.5% in the forensic data per case [10], the ability to perform such verification manually is quickly becoming impractical. Absent a proper specification of the forensic processing performed, there is no hope of automating the verification either.

• Scientific. The lack of common means to describe the processing makes it extremely difficult (and cost-prohibitive) to reproduce research and development work, and to perform tool testing and calibration. Current informal and indirect definitions of forensic computing have a very tenuous link to established computer science tools and methods. We believe that this is (at least partly) to blame for forensics remaining a relatively small research field, as researchers struggle to understand what the hard problems in forensics are.

• Educational. Currently, forensic education lacks even the means to express the equivalent of the “hello world” of forensic computing. Practically all courses on forensics we have examined start with now “classic” linear procedural definition (collection → extraction → analysis → reporting) which does not even mention the core computational problems being addressed. The hands-on alternatives are to: a) use an integrated environment with a GUI, which trains students which buttons to push but little about the computation being done; or b) perform the computations with low-level (open source) tools, thereby introducing a steep learning curve before anything useful could be accomplished. More generally, it is difficult to accumulate expert knowledge in a more abstract, tool-independent manner.

• Professional. In our view, one of the big limiting factors in forensic processing today is the failure of users to provide explicit tool requirements and hold vendors to them. This is due, in no small part, to the fact that most forensic analysts are not programmers in the same way software developers are, and have been unable to express their needs an appropriate technical form. As a result, tool development follows a somewhat backwards approach as vendors push new development—usually formulated as “supporting features x, y, z”—that are often disconnected from what analysts actually need.

The main contention of this discussion is that forensics, like other application domains, needs its own domain-specific language (DSL) as a critical component of the next generation of forensic systems.

One prime example of the impact a DSL can have on a field is the Structured Query Language (SQL), which decades ago became the standard for describing data and its processing in relational databases. The primary reason it became so popular is that it afforded its users the right abstractions to model the logical organization of their data entities, as well as an expressive mechanism to describe data query and manipulation.

One of the crucial aspects of SQL’s design is that it is a declarative language—it allows users to specify what should be done but relieves them from the burden of stating how. From a software development point of view, having a standard language serves as a formal specification for the runtime environment, and developers can focus on efficiently implementing the language primitives.

The rest of the discussion is organized as follows. Section 2 provides a brief overview of domain-specific languages; Section 3 discusses in more detail the motivation for building a forensic DSL; Section 4 provides a design sketch of the nugget language—an experimental prototype of a forensic DSL we are in the process of developing. Section 5 concludes the discussion with a summary of the presentation.
II. DOMAIN SPECIFIC LANGUAGES

A. Definition and properties

A domain-specific language is a computer programming language of limited expressiveness focused on a particular domain.

Following Fowler’s discussion [6], there are four important elements to this definition:

- First, this is a formal language whose statements get translated into machine code, and get executed on a computer. As with any programming language it is designed to make coding easier, but it is still a formal language with established rules and boundaries.
- Second, a well-designed DSL allows its users—typically, practitioners within the domain—to express the computation in a manner that feels natural and appropriate to a human; the language feels “fluent”.
- Third, unlike general-purpose programming languages, a DSL has limited expressiveness and is not designed to replace any of them. Rather, its purpose is to simplify development with respect to its domain.
- Forth, a DSL is focused solely on its domain and makes specifying computations in the domain substantially easier (than a general-purpose language). In other words, the DSL trades generality for simplicity, which makes it valuable to the user.

From a user’s perspective, the most important aspect is that the DSL contains language abstractions and constructs that directly model concepts of the problem domain. Indeed, when the mapping is done well, users of the language do not even think of themselves as programmers. For example, much of web development revolves around the HTML/CSS DSLs, but professionals in the area usually think of themselves as web designers/artists, not code developers; similarly, many business users have become very proficient in writing database queries with SQL without any special training, or programming carrier ambitions.

B. Example DSL: Apache Pig

Due to the short format of this discussion, we believe it would be more helpful to consider a case study of a relevant DSL rather than a comprehensive discussion on DSLs in general.

Apache Pig [2] is a dataflow language used to describe steps in the processing of large data sets. The specified computation is translated into (Java) MapReduce jobs that are run on a Hadoop [1] cluster. The aim is to allow users to focus on specifying the logic of the data processing and to relieve them from having to write low-level Java code to take advantage of the Hadoop environment.

Statements in Pig (Latin), the language of Apache Pig [8], work with relations; a relation is defined as follows:

- a relation is a bag;
- a bag is a collection of tuples;
- a tuple is an ordered set of fields;
- a field is a piece of data.

Each Pig statement specifies a relation transformation based on a previous relation and stores the result in a new variable. For example, the following script loads data (in three-column integer format), groups it by the first one and outputs a histogram:

\[
A = \text{LOAD } '\text{data}' \text{ USING PigStorage()} \\
\quad \text{AS} \ (f1:\text{int}, f2:\text{int}, f3:\text{int}); \\
B = \text{GROUP A BY f1}; \\
C = \text{FOREACH B GENERATE COUNT ($0);} \\
\text{DUMP C;}
\]

There are clear similarities between Pig and SQL in that they both allow the query of data organized in tables by means of selection (of rows) and projection (of columns). Yet, there are substantial differences in the way this is accomplished. In SQL, the SELECT statement combines the entire (potentially quite complex) query specification; in Pig, the same processing is given step-by-step with intermediate results stored in variables. Informally, the collection Pig statements can be viewed as “deconstructed” SQL.

Pig’s approach is a reflection of the fact that it is designed to facilitate the ad-hoc exploration of data sources. In such a scenario, the structure of the data is not known a priori, there are no strict relationships across the different tables, and there are no automatically managed index structures to take advantage of.

Below, we give a line-by-line annotated example, adapted from [9], to better convey the style of processing introduced by Pig. We will not elaborate on every single detail, such as the specific semantics of procedures like PigStorage(), NonURL(), etc., as these are tangential to our point.

Load log records of user queries:

\[
\text{raw} = \text{LOAD } '\text{query.log'} \text{ USING PigStorage()} \\
\quad \text{AS} \ (\text{user}, \text{time}, \text{query});
\]

Remove records if the query field is empty, or not a URL:

\[
\text{clean}1 = \text{FILTER \text{raw} BY \text{NonURL} (\text{query});}
\]

Change the query field to lowercase:

\[
\text{clean}2 = \text{FOREACH \text{clean}1 \text{GENERATE} \text{user}, \text{time}, \text{\text{ToLower} (\text{query}) AS \text{query}}; \\
\text{Extract the hour from the time field:}
\]

\[
\text{houred} = \text{FOREACH \text{clean}2 \text{GENERATE} \text{user}, \text{time}, \text{\text{ExtractHour} (\text{time}) AS \text{hour}, \text{query};}
\]

Compose the n-grams of the query:

\[
\text{ngram}1 = \text{FOREACH \text{houred} \text{GENERATE} \text{user}, \text{hour}, \text{\text{flatten} (\text{NGramGenerator} (\text{query})) AS \text{ngram;}}
\]

Get the unique n-grams for all records:

\[
\text{ngram}2 = \text{DISTINCT \text{ngram;}}
\]

Group records by n-gram and hour:

\[
\text{h_freq1} = \text{GROUP \text{ngram}2 \text{BY} (\text{ngram}, \text{hour});}
\]

Get the count (occurrences) of each n-gram:

\[
\text{h_freq2} = \text{FOREACH \text{h_freq1} \text{GENERATE} \\
\quad \text{\text{flatten} ($0$), \text{COUNT} ($1$) AS \text{count;}}
\]

5229
Use the **GROUP** operator to group records by n-gram only. Each group now corresponds to a distinct n-gram and has the count for each hour:

\[ u\_freq1 = \text{GROUP h\_freq2 BY group::ngram;} \]

For each group, identify the hour in which this n-gram is used with a particularly high frequency. Call **ScoreGenerator** to calculate a "popularity" score for the n-gram:

\[ u\_freq2 = \text{FOREACH uniq\_freq1 GENERATE flatten($0), flatten(ScoreGenerator($1));} \]

Use the **FOREACH-GENERATE** operator to assign names to the fields:

\[ u\_freq3 = \text{FOREACH u\_freq2 GENERATE $1 AS hour, $0 AS ngram, $2 AS score, $3 AS count, $4 AS mean;} \]

Select all records with a score greater than 2.0:

\[ \text{hi\_scores = FILTER u\_freq3 BY score > 2.0;} \]

Sort the remaining records by hour and score.

\[ \text{sorted\_hi\_scores = ORDER hi\_scores BY (hour, score);} \]

Store the results; the output file contains a list of n-grams with the following fields: hour, ngram, score, count, mean:

\[ \text{STORE ordered\_hi\_scores INTO 'hi\_scores.out' USING PigStorage();} \]

We could have probably written all of the above steps in one giant and unwieldy SQL query, but **Pig** provides a more incremental description of the computation. One could have some syntactic quibbles, such as some unnecessary verbosity, and the need to store every single intermediate result. However, we believe that **Pig**’s dataflow approach is, conceptually, an excellent match for what we need for forensic processing specifications. In particular, the fact that results are built incrementally, where each step is explicitly defined, and could be independently verified. Note also that the main unit of computation is a relation (or collection), not individual data items, which makes this suitable for mass data processing, and is a feature completely missing from the forensic tools of today.

The dataflow idea is also similar in spirit to the traditional style of **Unix** command-line scripting commonly employed in conjunction with open source forensic tools, such as **TSK**. The essential concept is to assemble the desired processing pipeline by stringing together a number of relatively simple tools. The shell environment provides for connecting the processing in a producer-consumer fashion either directly via pipes, or indirectly via files.

In their current forms, both **Pig** and shell scripts provide useful ideas, but their implementations do not directly address forensic analysis needs. In particular, **Pig**’s implementation is tied to **Hadoop** and lacks any forensic-specific abstractions that would help users relate to the language. Shell scripts have numerous expressive, performance, and platform limitations, and are not suitable for building large maintainable systems.

Having not found a satisfactory solution that could be directly employed, we set out to explore the construction of a new **DSL** with a focus on forensics.

### III. Language Design Considerations

**A. Who is our user?**

The obvious (short) answer is: the "typical" digital forensics investigator. The problem, however is that investigators come to (digital) forensics from two different backgrounds: a) technical, with programming skills and deeper understanding of computer systems; and b) non-technical (usually law enforcement), with more experience and understanding of crime and the legal system, but potentially no coding skills.

So far, practically all efforts at automating forensics have been targeting investigators with technical background. In our view, the prime user should be the non-technical one; in other words, an investigator with logical thinking and good analytical skills should feel at home using the language. This argues for a language that has a more declarative style, *a la SQL*, which focuses on defining *what* needs to be done, as opposed to *how* it should be accomplished. This is perfectly in line with purpose of DSLs to help bridge the gap between a specific domain and computing.

We should note that, by splitting the specification of the computation from its implementation, we also give the technical person an opportunity to shine by enriching the language, and improving the underlying implementation.

**B. Who not use SQL?**

One ostensibly low-cost idea would be to simply employ a more general, and already proven, data manipulation language, such as **SQL**, to accomplish forensic tasks. While such an approach has clear appeal, there are at least three major conceptual problems that make it unworkable.

First, forensics is different from practically all other computations in that it involves a critical *data recovery* phase that can be be quite complicated. We are not aware of any language that includes facilities for specifying such computations.

Second, using **SQL** means that we would need to define a specific, and *standard* data model—essentially, a set of database tables—to represent all cases. Considering the huge efforts and minimal success of much less ambitious standardization efforts (and the clear disincentives for vendors) we see no realistic way that this could be accomplished. Further, advanced **SQL** queries—which often involve complicated multi-table joins—actually *do* require good technical skills.

Third, the relational data model is a rather poor fit for expressing and querying graph relationships, such as a social graph, or dependencies among artifacts. This results in non-intuitive queries and, at scale, in poor performance and increased demands on the IT infrastructure. Even a cursory examination of the architectures employed by "big data" companies, such as **Google**, **Facebook**, and **Twitter**, shows that they approach this problem by matching the different data collection to different types of data stores in order to maintain performance. The only way to accomplish this in forensics is to abstract away the internal data representation—the exact opposite of what a **SQL**-centric solution would do.

The final step before we consider design solutions, is to find a more workable definition of forensics.
C. What is forensic computing?

Surprisingly enough, there has been very little effort to define the domain of forensic computing in terms that a computer scientist would recognize. This is likely due, in part, to the applied nature of digital forensics where a lot of the work originates in the field. Typically, the issue is avoided altogether by focusing on the end purpose of the computation—to support legal proceedings [5]—or by resorting to very fundamental mathematical definitions such as finite state machines and hypothesis testing [4].

While such approaches are valid and touch upon various aspect of forensic computing, we see them as the proverbial blind men trying to describe an elephant by feeling out different parts of the animal. As a result, there is a substantial gap between the models put forward, and the way practitioners think about forensic analysis. For example, the procedural “purpose”-driven definition is useful to a lawyer but it is quite incomplete—there are numerous uses of forensic analytical methods that are unrelated to legal proceedings—and still does not tell us what is being done. Connecting abstract mathematical models to practice in the field is even more difficult (and uninteresting) for most practitioners. Therefore, we use the following working definition as a starting point [11]:

The objective of digital forensic analysis is to reconstruct a chain of events that have resulted in the currently observable state of a computer system, or digital artifact.

The main benefit of this characterization is that we can now focus specifically on the question of how forensics is performed by fleshing out the details of the definition. In particular, there are two aspects that need elaboration: a) how is the “state of the system” is defined; and b) how is the “chain of events” reconstructed.

Note that, if we remove the words computer and digital, we end up with a generic definition that could easily be applied to forensic analysis in general (in case of robbery, plane crash, Van Gogh forgery, etc.).

The state of a computer system consists of the state of its hardware, and its software. The latter can be usefully split between the state of the operating system (OS), and that of any applications that have been executed. In general, the state of any non-trivial piece of code is quite complicated and difficult to capture in its entirety; fortunately, most of it is transitory and, ultimately, irrelevant to the forensic inquiry.

Thus, we define the (forensically-relevant software) state of the system as the observed state of relevant OS and application data structures. The question of relevancy is non-trivial and, in the general case, is tied to the specific interest of the inquiry. Relevancy is often established by empirical and/or reverse engineering methods; understanding what state is relevant to an inquiry is a big part of being a forensic expert.

In other words, the first step to a sound forensic analysis is to identify relevant data structures from the evidence sources, and extract their state. These often include a list of files, processes, registry entries, etc., as well as the content of specific files (which are the external representation of OS/application state).

The second step is to interpret the observed state based on known causal relationships of the form prior state \(\rightarrow\) computation \(\rightarrow\) observed state. Clearly, the most valuable relationships are those that yield a unique outcome, allowing us to step back in time and establish causality retrospectively. For example, MS Windows creates a record the first time a USB device is attached to the system containing the unique identifier for the device. Consequently, finding such a record allows us to conclude that a specific device was attached to the system.

More generally, the primary deductive tool of forensics is differential analysis [7], which allows us to build a knowledge base of state transitions that are of significance, and gives us a framework for approaching new situations.

The goal of forensic software tools should be to provide the appropriate abstractions that allow analysts to fluently express the deductive process as a series of computations. Yet, the analyst should not have to provide detailed instructions as to how the computation is to be performed, especially for routine processing.

D. What are the basic abstractions specific to forensic computing?

Experience shows that successful ideas in computing start simple and grow over time. Therefore, our immediate objective is to identify a minimalistic set of abstractions that would allow us to experiment with an early prototype and receive feedback from experts.

**Primitive data types.** Like any other language, we need a few basic data types. In our case, we start with a collection of signed/unsigned integer types, which explicitly specify their size in bits: `int8/uint8`, `int16/uint16`, `int32/uint32`, `int64/uint64`.

We also need the concept of the string, which is an array of characters that may be encoded in any number of ways, such as ASCII, Unicode, and base32/64/85. For completeness, `nugget` also has a (double-precision) float type.

**Domain-specific data types.** In forensics, there are a number of commonly used data types that need to be first-class concepts. These include, time values (`datetime`) and various types of hash values. These allow the runtime to provide sensible default behavior in common comparison, sorting and filtering scenarios.

**Collections.** We use the terms collections where Pig uses relations; collections are more flexible in that we do not expect every element to have the exact same structure. One can think of them as being schemaless in the way most NoSQL data stores are. Scala, our implementation language, has an extensive collections library that allows common collective operations, such as filtering, ordering, etc., to be expressed succinctly (in the tradition of functional programming).
In addition to the base constructs, we expect the language to accumulate a substantial library of utility methods similar to those found in Pig. More importantly, these would be easily extensible by the user, to enable custom processing flexibility where needed.

IV. NUGGET: AN EXPERIMENTAL DSL FOR FORENSICS

In this section we sketch out our early prototype work on Nugget—a DSL for security and forensic applications. We should emphasize that this is early exploratory work, and not complete prototype. Therefore, we choose to introduce it by example, as the complete design has not yet been formalized. Clearly, such formalization would be a precondition for any efforts to put the language to practical use both in the field, and for educational purposes.

We draw on design ideas from Pig and from Unix shell processing, with appropriate syntax simplification and specialization. Nugget is a data flow language that is adapted for the needs for forensics. Unlike Pig, which seeks to be generic in its processing (a la SQL), we want a language that is entirely focused on the domain.

Conceptually, a data flow computation starts with a source stream (a series of data items) that is passed on through a pipeline of filters. Each filter transforms the data items and passes them on to the next stage. This is very much in the spirit of shell processing, with the notable exception that the language runtime has semantic knowledge of stream content. At each stage, an intermediate result can be remembered by assigning it to a variable; once computed, a result is immutable.

A. Nugget by Example

Data could either come preprocessed from a file, or could be extracted from a raw source using an appropriate parser. Let us consider the case of extracting files from a disk image:

```
f = "file:target.dd" extract files
```

The interpretation is straightforward—the raw data source (file named “target.dd” containing the disk image) is passed on to the extraction parser, with the files extraction option. The result is a collection of all regular files on the image, referenced by variable f.

We should be quick to point out that the above statement would not actually create a copy of all the files; only a copy of the filesystem metadata would be made. The next thing we want to do is filter down the set to include only PDF files created since Jan 1, 2014:

```
pdfs = f | "*.pdf", ctime > "01/01/2014"
```

The pipe command "|" streams the content of f, which is a sequence of key-value pairs of file attributes, such as name (file name), ctime (creation time), content (actual file content), etc. Since no filter is given explicitly, the default select filter is run on all file attributes, with the exception of content and all files that pass both criteria would be included in the result. In the next step, let’s hash the content using MD5:

```
hashed = pdfs.content | hash md5
```

In this case, the content of the files in the collection is explicitly invoked so it is extracted from the original image (via a read-only mount), and passed onto the hash filter with the md5 option. The hash filter will compute the requested hash value, and will add a new attribute called md5 to each element in the stream. In the last step, we’d like to find all the known files:

```
known = hashed | join "file:known.md5"
```

The runtime will take a cue from the file’s extension, as well as the collection’s attributes to automatically perform the join operation on the md5 attribute and the corresponding column in the reference file.

Unless the intermediate results are of further interest, we could significantly shorten the script:

```
pdfs = "file:target.dd" extract files | "*.pdf", ctime > "01/01/2014"
known = pdfs.content | hash md5 | join "file:known.md5"
```

It is worth noting that, like in Pig, no computation actually takes place until there is a command that explicitly requires the results, such as to print a textual representation on the terminal, show a graphical representation of the result, or save the result to a file (in the preferred format):

```
save known to "known_pdfs.json"
```

This lazy evaluation approach gives the runtime the maximum amount of information on data dependencies. Much like a compiler rewrites code during translation to optimize performance, Nugget omits attributes mentioned but not actually used in the computation, and distributes the evaluation across all available computer resources.

Projection, another core function from database algebra, is performed using the keyword as and specifying the attributes to be included either by name, or by position number:

```
fnames = f as (name, size)
fnames = f as ($2, $5)
```

The system can sort and group the results (like SQL) and can also produce histograms:

```
histo = fnames histogram by size autobin
```

The extraction parser could be instructed to do a complete run and, in addition to files, extract all the (p)artitions, and registry (h)ives. The result is a tuple of collections:

```
(p,f,h) = "file:target.dd" extract all
```

Extraction is an abstract operation and the actual implementation chosen by the runtime depends on the target. In the above examples, it would be the TSK [3] as we a working with a file system image. If the image were a RAM capture, Nugget would use Volatility to do the parsing:

```
p = "target.ram" extract processes
```

However, if we used a host name, as opposed to a file, a live capture mechanism would be used (if one is available):

```
p = localhost extract processes 
as (pid, ppid, command)
```

5232
The ability to work with different data sources is a prerequisite to supporting a variety of applications for the language from traditional forensics to security monitoring and incident response.

B. Implementation notes

The current prototype of nugget is built as an internal DSL in Scala. The latter is a statically-typed language designed to allow developers to build flexible and expressive language constructs tailored to their problem space. In other words, it is the perfect environment for developing an extensible DSL.

An important benefit of using a statically-typed language is that we can provide a more user-friendly environment that provides auto-complete and real-time syntax feedback. The Scala language itself is proof that the vast majority of benefits widely attributed to dynamic language can be achieved without giving up the benefits of static type checking.

Scala is a JVM-compatible language, which means that the compiler produces Java bytecode and all the standard Java facilities are available. This makes it relatively easy to employ a number of existing solutions; however, we envision a runtime system that is built entirely around services and is language-agnostic. Looking forward, a service-oriented architecture is essential for deployments in cloud environments where the available resources can be matched with the demands of the processing tasks.

One aspect we did not explicitly discuss in terms of motivation is the need for scalable forensic tools that can match the rapid growth in data volume. At first glance, it may not be clear how a high-level DSL can help with the problem of scalability. The key insight here is that by raising the abstraction level we relieve users from the need to explicitly optimize the computation—a task they are inherently ill-equipped to perform. A nugget script provides a description of all the data sources and their dependencies, which enables the runtime to make intelligent choices on how to parallelize as much of the computation as possible. Contrast this with the user having to put together a processing pipeline out of bits and pieces—even if all the components are individually optimized, the computation as a whole will suffer from communication bottlenecks that serialize the execution of the components. Nugget allows users to think and describe the computation in serial steps, but the execution engine is free to optimize for performance.

V. Conclusion

Most digital forensic tools fall into one of two categories—large, closed (commercial) integrated environments with GUI interfaces, and collections of open (often very capable) command line tools. In either case, there is no abstract mechanism to specify the computation in a tool-independent fashion.

To address the problem, we introduced nugget—an early prototype of a domain-specific language focused on security and forensic data processing, and we model such processing as a series of extraction and filtering operations on operating system and application data structures. The main purpose of the language is to provide an appropriate level of abstraction for describing such processing that is simple and intuitive to use by analysts, but also serves as a de facto requirements specification for tool developers.

The nugget language and runtime environment are designed to be an extensible infrastructure that incorporates both existing and future processing tools. We seek to incorporate existing capabilities, not replicated them; we seek to provide a higher level interface to use them, as well as supply a platform to integrate capabilities across tools. We want to relieve analysts from the burden of irrelevant details and resource optimization and let them focus on their case.

In the near term, we expect to complete a stable prototype that provides the first official version of the language and, at a minimum, integrates TSK [3] and Volatility [12] capabilities into the runtime. In the medium term, we plan to push the language development towards the use of big data tools for analytical purposes.

References