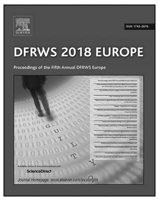
Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2018 Europe — Proceedings of the Fifth Annual DFRWS Europe

Nugget: A digital forensics language

Christopher Stelly*, Vassil Roussev

GNOCIA, Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
Digital forensics
Nugget
Domain specific language

A B S T R A C T

One of the long-standing conceptual problems in digital forensics is the dichotomy between the imperative for verifiable and reproducible forensic computations, and the lack of adequate mechanisms to accomplish these goals. With over thirty years of professional practice, investigator notes are still the main source of reproducibility information, and much of it is tied to the functions of specific, often proprietary, tools.

In this work, we discuss the design and implementation of a *domain specific language* (DSL) called *nugget*, which aims to enable the *practical* formal specification of digital forensic computations in a tool-agnostic fashion. The core idea of DSLs, such as SQL, is to create an *intuitive* means for domain experts to describe *what* computation needs to be performed while abstracting away the technical means of its implementation.

In the context of digital forensics, *nugget* aims to address the following requirements: 1) provide investigators with the means to easily and completely specify the data flow of a forensic inquiry from data source to final results; 2) allow the fully automatic (and optimized) execution of the forensic computation; 3) provide a complete, formal, and auditable log of the inquiry.

Introduction

The first steps towards the professionalization of digital forensic investigations in the US date back to the mid-1980s, and stem from the passage of the first legislative acts on computer crime (Comprehensive Crime Control act of 1984; 18 US Code), and the establishment of FBI's *Magnetic Media Program* (Garfinkel, 2010). Importantly, the impetus to examine digital evidence came from law enforcement concerns and was initiated by *investigators* with technical knowledge, rather than software engineers, or computer scientists.

Over time, with the enormous growth in data volume and complexity, it became necessary to develop specialized tools to support the process, including a number of commercial tools. Although these are not inexpensive, their development has largely followed an ad-hoc "feature accumulation and GUI beautification" approach. Given the economic incentives, it is entirely understandable that each vendor is working to attract and lock in as many customers as possible to their system. The problem is that this requires near blind trust in the deployed systems, and offers no ready means to perform third-party verification of the results; this becomes increasingly unacceptable as the volume and importance of the examined data continue to grow rapidly.

The investigator is increasingly becoming a tool operator and is ever more detached from the (implementation of the) methods used to process the evidence. This is the inevitable result of the build-up in complexity of the target system; however, the lack of proper means to specify and verify the correct forensic system behavior is a fundamental issue that will continue to grow until addressed directly. The first step in this process is to disentangle the *specification* of the investigative inquiry from its technical *implementation*. The inquiry is inherently case-specific, and is the responsibility of the forensic analyst; the implementation is the responsibility of researchers and/or software developers, and there are (almost always) multiple approaches to how it can be carried out. By having a well-defined query interface in between, it becomes possible to directly compare the results of multiple implementations, and to establish the ground truth. For a quick illustration, consider the following example:

Listing 1. Example *Nugget* query.

```
recentpdfs  = "file:target.dd" | extract as
   ntfs[63,512] | filter ctime > "01/01/2017"

known = recent_pdfs.content | sha1 |
   join "file:known.sha1"
```

* Corresponding author.
E-mail addresses: cd@stelly.org (C. Stelly), vassil@roussev.net (V. Roussev).

https://doi.org/10.1016/j.diin.2018.01.006

Even at first sight, most forensic professionals would readily recognize the above query as an instance of *known-file filtering*; in this case, it is applied to all `pdf` files extracted from the `target.dd` source, and created after Jan 1, 2017. The main points here are:

i. the domain expert did not need to learn a general purpose programming language in order to understand the intent of the query (and could quickly learn to write similar ones);
ii. this is a formal specification that can readily be translated into executable code;
iii. the query only specifies *what* needs to be done, and not *how* it should be performed; there are numerous possible implementations, including ones that employ the resource of a computer cluster, or a (private) cloud service;
iv. the query itself unambiguously documents the forensic process and allows for automated testing, verification, and reproduction of the results.

In other words, *nugget* seeks to do for forensic computing what SQL did for relational databases: establish a standard query interface that is complete and intuitive enough for domain experts to readily understand, while also providing a formal specification of the computation that needs to be carried out. SQL allowed for numerous competing implementations to co-exist, which allowed for fast development, optimized execution, and autonomous GUI development.

Concept Overview. Before we dive into specifics, it is important to place this effort in a larger context and recognize that the *nugget* DSL is one component of a larger research effort to address holistically a set of issues related to reproducibility, performance, and scalability. As shown in Fig. 1, the language is an interface between the UI layer (textual, or graphical) and the runtime environment & resource manager. The DSL presents a unified means to execute forensic computations (using the available set of tools), organize them in processing pipelines, and store/return the results as needed.

The language runtime maps the abstract representations of an operation, such as the extraction of a list of processes, or the hashing of a file, to an actual command to be invoked on the selected target. This is driven by user specifications, and allows the incremental extension of *nugget* with new capabilities; in fact, the entire current language implementation is specification driven. The resource manager is tasked with scheduling the computations on the available resources, ensuring their successful execution, logging all operations performed, combining the results (if executed on a cluster) and returning the results of the computation.

This architecture disentangles the concerns of a) specifying the computation, b) mapping it to the available tools, and c) scheduling it on the available hardware resources. This layered approach is conceptually different from the two options currently available to analysts: 1) a bundled (black) box of tools with a point-and-click interface (primarily, commercial vendors), or 2) a bag of tools and components from which the analyst must craft (i.e., code) together the desired solution (open source tools). None of these offer a solution that adequately addresses user needs and cost concerns, and none support standardized independent testing and provable reproducibility.

One of the principal problems in digital forensics is the lack of clear means for users (forensic analysts and lab managers) to communicate functional and performance requirements to vendors. The main point of *nugget* is to solve this problem by allowing analysts to directly specify queries they can reason about, and to demand responsive solutions; it allows users to directly compare alternatives, and creates a best-of-breed competition among vendors. Conversely, a formal interface allows developers to have specific targets, and to better understand the needs of their customers.

Contributions. The work presented here is based on the ideas presented in (Roussev, 2015); our main contributions are:

i. We provide an actual language implementation based on an *external* DSL, which allows for implementations in different programming languages to be incorporated (the original position paper mentions an internal one, which was never published);
ii. We provide an integration framework, based on containers, which allows for a) existing specialized tools, such as command-

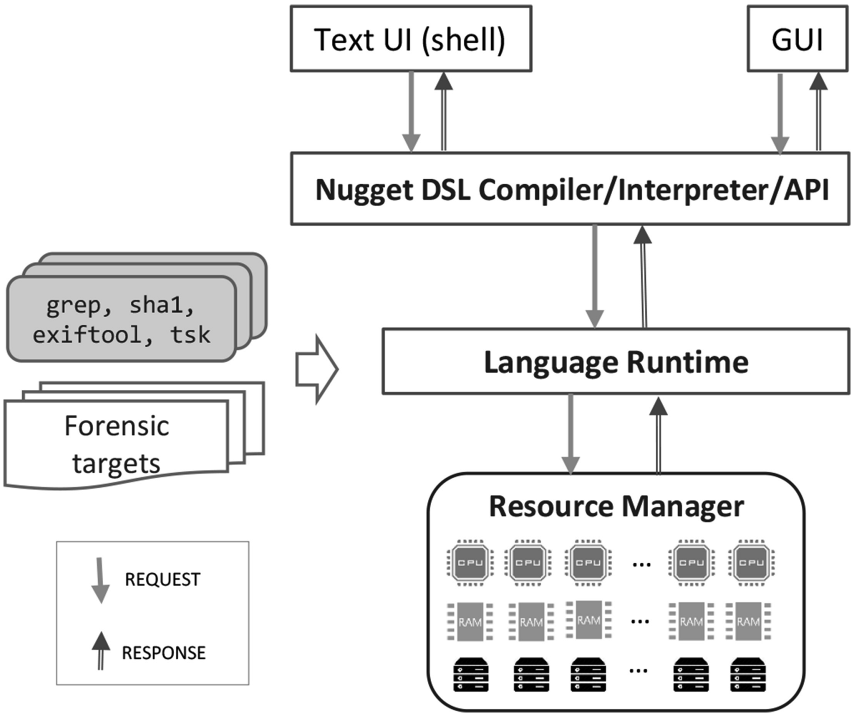


Fig. 1. Layered forensic runtime architecture.

line utilities, to be seamlessly integrated into the forensic process; and b) the language to be dynamically extended by providing descriptions of the available components.

Related work

In this section, we briefly describe the most closely related work to our own. Specifically, we offer a brief characterization of different models of forensic computations, as well as work on DSLs relevant to our discussion.

Models of forensic computations

There is no shortage of efforts to formally describe the digital forensics process. Some of the more influential ones have been Carrier's *hypothesis testing model* (Carrier and Spafford, 2006), Garfinkel's *differental analysis* (Garfinkel et al., 2012), and Gladyshev's *finite state machine* (Gladyshev and Patel, 2004). These are all conceptually valid, and bring interesting ideas from other domains (of mathematics). However, they start at the lowest level of abstraction to describe the computation and, ultimately, do not lead to practical means of specifying the computation.

At the other end of the spectrum, Roussev has proposed [Roussev, 2016, Ch. 3] that a cognitive task model developed to describe the work of intelligence analysts (Pirolli and Card, 2005) could be directly adopted to describe the cognitive tasks performed by forensic analysts. Although there are clear benefits to considering the problem from a cognitive perspective (especially for the purposes of usability), the resulting description is not formal and does not address the concerns of integrity and reproducibility.

In between, we find a large number of procedural models that deal with specific investigative scenarios, such disk acquisition, Android forensics, social media analysis etc. These are, in effect, efforts to establish best practice guides for practitioners (e.g., (Scientific Working Group)). They are more specific than the cognitive approach, and are actionable, but fall well short of being formal, generic, and reliably reproducible.

Our work aims at the midpoint between best practices and purely mathematical models: we define a formal model that works at the same level of abstraction as the analyst (like best practices), but leads to an unambiguous computational description (Fig. 2).

Forensic DSLs

The idea of DSLs for the purposes of digital forensics has been explored by several prior systems. Most explicitly, the *DERRIC* project at the *Netherlands Forensic Institute* introduces a language (Bos and Storm, 2011) to declaratively specify data structures, allowing for data processing tools to execute upon multiple variants of data types. Similar ideas appear in the design of several tools, such as *binary templates* supported by *010 Editor* (SweetScape Software Inc) and *vtypes* in Volatility [Ligh et al., 2014, p. 51]. Volatility also defines a number of *Python* objects with common functionality and provides a framework for extending the functionality of the system. Although somewhat constrained, it can be viewed as an internal DSL built for the purposes of memory forensics.

The main difference between *nugget*and prior efforts is that it seeks to be general and extensible enough to extract and query all common types of digital evidence and to describe the flow of their processing from source to end result. We seek to address the *entire* domain, not just parts of it. At this stage, *nugget* does *not* attempt to abstract away the source data representation; our goal is to integrate existing tools (which already understand the data) rather than define abstract layer to aid the development of generic tools (This would be a natural extension at a later point.)

Data query DSL: Apache Pig

In the original position paper (Roussev, 2015), *Apache Pig* was cited as the immediate inspiration for the proposed design. Without attempting to be exhaustive, we offer a brief description by example.

Apache Pig is a dataflow language designed to describe the incremental steps in the processing of large datasets. Its primary users are data researchers and programmers, so it is designed to support interactive exploration. The actual computation is translated into (Java) *MapReduce* jobs that execute on a *Hadoop* (Apache Hadoop) cluster. The inspiration for *Pig* comes from *Sawzall*—a similar language (Pike et al., 2015) developed at Google to serve as an abstraction layer over the company's *MapReduce* infrastructure.

The basic abstractions in *Pig* are *fields*, *tuples*, *bags*, and *relations*:

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

Each *Pig* statement specifies a relation transformation based on a previous relation and stores the result in a new variable. For example, the following code (Roussev, 2015) loads data in three-column format, groups it by the first one, and outputs a histogram:

```
A = LOAD 'data' USING PigStorage()
    AS (f1:int, f2:int, f3:int);
B = GROUP A BY f1;
C = FOREACH B GENERATE COUNT ($0);
DUMP C;
```

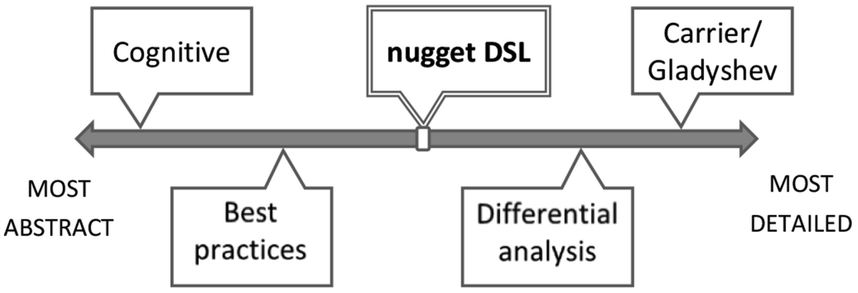


Fig. 2. Nugget vs. Other models of forensic computation.

Although both *Pig* and SQL are both data query languages, there is a notable difference between them. Unlike SQL, the *Pig* query is constructed step by step. This approach is a reflection of the fact that the language is designed to facilitate the ad-hoc exploration of semi-structured data sources—a scenario quite similar to a lot of digital forensic analyses. It is also worth noting that the main abstraction is the *collection*, which can be manipulated as a first-class object.

The case for digital forensic DSL

As per Fowler (2010), *a domain-specific language is a computer programming language of limited expressiveness focused on a particular domain*. The following are the critical components of this definition:

i. *Formality.* A DSL is a formal language, with established grammar and semantics, that is translated into *executable* code.
ii. *Fluency.* A good DSL allows its users—practitioners within the domain—to express the computation in a manner that is “fluent”; i.e., it feels natural and appropriate to a human expert.
iii. *Limited expressiveness.* A DSL is *not* designed to replace a general-purpose programming language; its purpose is to simplify development *with respect to its domain.*
iv. *Ease of use.* A DSL is focused entirely on its target domain and makes specifying computations in the domain *substantially easier* than a corresponding solution in a general-purpose language. That is, the DSL trades generality for simplicity, which makes it valuable to the user.

In sum, a DSL gives domain experts the constructs necessary to describe a problem, or solution, succinctly and efficiently, and abstracts away all (or most) references to the actual implementation. End-users can accomplish significant, complex tasks with a small number of keywords and phrases. This common vocabulary makes the language feel more natural to end-users, resulting in a lower learning curve (Ghosh, 2010).

Examples of popular and robust DSLs are plentiful: SQL is the *lingua franca* of the database world (even after a decade of the “noSQL” movement); the runaway success of the web is, in part, due to HTML and CSS – two DSLs whose users do not even perceive writing in those languages as coding. Unix/Linux shell scripts have been a mainstay of system and network administration, and have been in widespread use by forensic analysts since the very beginning.

Over the last two decades, we have seen an accelerated trend towards the development of programming languages, and domain-specific ones, in particular. This is, in part, driven by the needs for much higher levels of automation in large-scale (cloud) environments, as well as the necessity to increase the level of reliability by means of formalizing the state and state transitions of complex systems. For example, in the area of automated configuration management, we have seen the rapid adoption of *Puppet*, *Ansible*, *Chef*, and *Salt*, along with container/VM configuration languages by *Docker* and *Vagrant*.

Why develop a forensic DSL now?

Viewed in a narrow light, development of a DSL is a costly endeavor. It requires significant knowledge of both programming techniques and of the particular domain in question. Further, when a DSL reaches a large user base, there should be an expectation for upkeep costs as bugs are fixed, training materials are produced, and new features are introduced (Mernik et al., 2005). Finally, performance *could* be a concern as it is another layer of computing. In this section, we briefly justify the need for a comprehensive DSL for digital forensics.

Bridge the semantic gap. Generally, a well-designed and efficiently implemented DSL can dramatically expand the number of users that can autonomously solve problems within their domain of expertise, especially problems which previously presented significant technical hurdles. This major usability gain comes from the fact that DSLs are concise (Ghosh, 2010), which reduces the semantic distance between the program and the problem. In other words, the language allows users to employ abstractions using natural terms and phrases.

The reduction of the semantic gap is a pressing concern in digital forensics as investigators work with ever more complex targets and cannot be expected to understand *in depth* the technical implementation of the tools utilized. They still need to understand the methods conceptually, and be familiar with the reliability and error characteristics of the methodology, but it is highly unrealistic to expect the average forensic analyst to be an expert researcher and code developer.

Improve reliability and reproducibility. As already discussed, two broad categories of (digital) forensic tools have evolved – (mostly proprietary) integrated forensic environments that provide a point-and-click interface, and a large collection of (mostly open source) specialized tools that address specific problems. Each category presents different problems: integrated tools provide few, if any, means to log and verify individual steps in complex scenarios, which makes it impractical to test and validate them; specialized tools provide better visibility but require custom integration, which is both costly from an operational perspective, and leads to the development of bespoke environments that are difficult to test in a standardized, automated manner.

A widely supported DSL would allow for a unified means to specify, log, and systematically test both individual forensic functions and integrated implementations. It also helps address the traditional tension between proprietary implementations and the need for testing and the need to independently establish the validity of tools via third-party testing. In this scenario, a vendor needs only to support a standard means of specifying the query, a (simple) standard format of returning the results, and a standardized log format. A community standards body, such as NIST, could perform independent testing which would go a long way towards alleviating reliability and reproducibility concerns.

Facilitate tool integration. As conceived, the specific proposed DSL solution – *nugget*– and its specification-driven implementation, provides an additional benefit in that it allows the integration of a group of tools to accomplish the necessary task. Clearly, this would greatly benefit the open source toolset, and would combine the advantages of open code with those of an integrated environment. However, it also allows the incorporation of proprietary tools in the forensic environment; experience from other domains where open source tools are a major presence shows that we still need vendors to develop more advanced/specialized components.

Integrate big data analytics & AI. Looking slightly ahead, we can expect a growing fraction of the evidence to be sourced from online services rather than physical devices [Roussev, 2016, Ch. 6]. This shift will all but eliminate many of the issues related to acquisition from physical devices, and will bring to the fore the critical need to employ big data techniques and resources to analyze the huge volume of data. As pointed out in (Roussev et al., 2016), most current tools' implementation is tied to the filesystem API; a DSL will provide a seamless transition by allowing the tool implementation to be replaced without modifying the forensic queries.

Volume growth will also necessitate the utilization of machine learning/AI methods to raise the level of abstraction of the analysis. For example, we can expect the use of computer vision systems to index and analyze the content of photo and video artifacts. A DSL can seamlessly integrate such advances by incrementally expanding the language.

Streamline education and training. Just like SQL allows the (relational) database-related education and training courses to provide meaningful skills without understanding the database engine implementation, we expect a language like *nugget* to provide the medium for training competent investigators without overwhelming them technical details. Over time, we would expect that *basic* investigative functions that are entirely sufficient for educational purposes to be available in an open source format, whereas a more advanced processing would likely require more specialized training.

Utilize mature DSL development tools. There has never been a better time to develop a new DSL; after four decades of evolution, language development tools are mature and reliable, and modern IDEs provide real-time help and feedback to users (based on the formal language specification). The latter considerably reduces the length and steepness of the learning curve.

Nugget DSL

In this section, we describe the design and development of *nugget*. To facilitate the introduction of the language, we preface it by providing a brief introduction to DSL construction.

Internal vs external DSL

Domain specific languages can be classified as *internal* or *external*. *Internal* languages are an extension of their host language — that is, they add constructs to an existing general-purpose language and are supported by the host language toolset. In particular, all input is parsed with the host language's constructs and results in the generation of code in the host languages. One popular example of an internal language is *Rails*, a web development DSL whose host language is *Ruby*.

Internal language advantages include the ability for end-users to call upon the full power of the host language. Modern languages, such as *Scala* (Odersky et al., 2016) have advanced built-in support for developing internal DSL. The major disadvantages are that a) valid syntax for the DLS *must* conform to the syntax and semantics of its host language, and b) integration with tools written in other languages is not readily available.

External languages are completely independent insofar as lexical analysis, parsing, compilation, and code generation are concerned. The advantage here is that developers are free to create (and extend) their own syntax and semantics; however, the disadvantage is the loss of direct access (from the DSL) to the host language's features.

Overall, internal DSLs, which are faster to develop, are appropriate where the scope is expected to be limited, and tight integration with the host language is a requirement. External DSLs are needed whenever a more general solution is sought, and one that is implementation language independent.

One illustration of the tradeoff is the development of *Puppet*, which originally had a *Ruby DSL*, which was later abandoned in favor of an external solution (Puppet Labs, 2015). Our work also started as an internal DSL; however, we quickly reached its limitations and switched to an external DSL that can be dynamically extended.

Nugget concepts

The basic data unit of *nugget* are collections of objects in the style of *JSON*; each object consists of a series of key-value pairs. Values are of several familiar primitive types, such as 8/16/32/64-bit integers, strings, and dates, as well as several specialized data types, such as binary/hexadecimal/base64 strings and standard kilo/mega/giga/ … notation for data units.

With the exception of output statements, each line in the code is a variable assignment. The right-hand side starts with a data source, which is either a named external source (such as a disk image) or a variable name (a reference to an existing collection). The pipe symbol, "|", serves to concisely connect the multiple operations in a single flow statement.

There are four types of operators that are used to describe the computation: *extractors*, *filters*, *transformers*, and *serializers*. We use the example code in Listing 2 to concisely explain their intended use.

Listing 2. NTFS file extraction, filtering (by size), and hashing.

```
1  files = file:target.raw |
2      extract as ntfs[63,512]
3  big_files = files | filter size > 1M
4  hashes = big_files.content | sha1, md5
5  big_files = big_files |
6      drop ctime | add hashes
7  print big_files
```

The main function of *extractors* is to shield the rest of the system from the particulars of the data format and method of ingesting the source. Extractors are operations that take as input a data source, such as disk/RAM image, and produce as output collections of data items, such as files, processes, packets, etc. In other words, extractors parse raw data input and produce entities with known (to the system) logical structure. In this terminology, data carvers are considered extractors, and so are operations that obtain the data via an API to a live system (such as a running kernel, or a cloud service). Reading from a supported forensic container also falls under the category of extraction.

In our example, we use an NTFS extractor (from the Sleuthkit), which parses a raw disk partition and extracts the filesystem metadata. In the specific case, we supply two additional parameters, 63 and 512, which provide the starting block and block size, respectively. Implicitly, each object is created with a set of known attributes, such as *name*, *size*, and *ctime* (creation time). One special attribute, *content*, references the data content of the file. To avoid unnecessary I/O operations the content is retrieved *only* when explicitly required.

Filters are data reduction/expansion operations that manipulate the result set by means of removing (filtering out) objects, and adding/dropping of object attributes. Line 3 of the example query filters out all files 1 MB in size and smaller (the condition specifies which objects should be kept in the result; the keyword "filter" is optional). Line 6 instructs the runtime to remove the *ctime* attribute (mostly for illustrative purposes) and to add two more attributes, *sha1* and *md5* containing the eponymous hashes of the content.

Transformers are functions that produce output, such as a hash value, for each object in the input collection. On line 4, two values (a tuple) is produced based on the content of each file in the input set.

Serializers are functions that produce an external representation for a collection; for example, *print* yields a textual representation suitable for shell environments. Different versions of *save* can produce json/xml output suitable for storage. Subsequent work will integrate specialized evidence containers like AFF (Garfinkel et al., 2006; Cohen et al., 2009).

Nugget delays execution until resolution of variables is required, exhibiting *lazy evaluation*. This allows our analysts to lay out their logical sequence of steps without concern for the optimality of execution time. Later iterations of the implementation will feature query optimizations, much like those supported by SQL engines.

Nugget's grammar

The standard means of describing the grammar of formal languages is the *Extended Backus-Naur Form* (EBNF) (Wirth, 1977; Scowen, 1998). The essential concepts are those of *terminal symbols*, such as the literal numerals "1", "2", and "3", and a *non-terminal production rules*, or sequences. Production rules, often nested or chained together, govern the valid sequences of terminal symbols, and thus the legal syntax of a language. For instance, the following definition describes the syntax for a simple calculator capable of addition and subtraction.

```
Operation: Number (Symbol Number)+;
Symbol:  "+"  |  "-";
Number: ('0'..'9')+ ('.' ('0'..'9')+)?;
```

Nugget employs a context free grammar described with ANTLR's version of the EBNF notation. Each of the statements in our sample code are *assignment* statements, which in EBNF looks as follows:

```
assign: (ID '=' STRING ('|' nugget_action)*   |
             ID '=' ID ('|' nugget_action)*  );

ID : [a–zA–Z]+;

nugget_action :
    'filter' filter_term (',' filter_term)*;
    'extract' asType   |
    'sort'    byField  |
    'sha1'             |
    ...
```

The *assign* rule states that an assignment can occur to an *ID* from either a literal string (used for references to local files) or another *ID*, followed by an action. In practical terms, *ID*s are limited to valid variable names. *Nugget_action* is an optional and repeatable construct following a required |.

Similarly, the *nugget_action* rule defines the syntax for a variety of actions (where an *action* can be a transformer, filter, or an extractor). In conjunction, these quickly allow for complex queries to be described (see Fig. 3).

It is impractical to attempt to exhaustively explain every *nugget* clause; however, it is important to note that we define valid actions *within* the grammar itself. That is, if we attempt to provide input to *nugget* with an undefined action, there will be a syntax error at the *parser level*. This is a stricter scheme that allows early error detection; the alternative is to allow any valid string, and leave all syntax checking handling to the consuming application.

This design decision was made deliberately – embedding valid actions within the grammar itself makes extending the language more complicated, which runs contrary to one the design goals of *nugget*. However, the benefit of this approach – error handling at the parsing level, and syntax checking *prior* to compile time – are more important from a usability perspective. Further, syntax checking can be extended to support code completion – a critical feature which has become common in development environments.

To retain relative ease of extensibility, we provide users with an automated build tool that allows them to rebuild the language based on simple function specifications, as illustrated later in our discussion.

Generating language constructs with ANTLR

Nugget relies on *ANTLR* for lexical analysis, parsing, and building an abstract syntax tree (AST), which is "walked" to execute indicated operations. *ANTLR* (ANother Tool for Language Recognition) (Parr, 2013) is an open source *parser-generator*. It is capable of taking an input grammar and producing the appropriate lexical and parser functions necessary to consume legal inputs. ANTLRv4 is capable of generating these functions in a variety of output languages: C++, C#, Go, Java, Python, JavaScript, and Swift.

The AST. Like other programming models, the end result of the lexing operation is an abstract syntax tree (AST). ASTs are a useful structure which allow compilers to walk across the tree. As the tree is walked (from left to right), it recursively descends into children nodes, executing corresponding functions within the application source code. For example, the AST representing our sample *nugget* code listed in Listing 1 is shown in Fig. 4.

Nugget runtime

Recall that one of the primary design goals of *nugget* is to provide a common interface for interaction with a variety of forensic tools. The runtime integration of forensic tools is based on our prior work in (Stelly and Roussev, 2017), which employs a combination of remote procedure calls (RPC) and Linux containers via *Docker*.

Containers and Docker. Containers provide encapsulation of a process' runtime by providing access to the set of resources—CPU cores, RAM allocation, file systems, and networking—needed to perform a computational task. All containers share a common OS kernel but, by default, are isolated from each other; it is also possible to setup sharing of resources where needed, e.g., software installations. Generally, containers have a much smaller resource footprint than full-stack VMs, and the overhead to startup/shutdown a container is comparable to that of a regular process.

For our proof-of-concept integration, we have built three separate tool-specific containers: a *Sleuth Kit* container for hard disk forensics, a *tshark* container for network forensics, and a *Volatility* container for memory forensics. We should emphasize that it is very easy to both containerize existing tools and to integrate them into *nugget*. Thus, the specific containers we use here can be replaced by similar tools; alternatively, multiple versions of the computation could be run in parallel (e.g., a *Volatility* container and a *Rekall* container) to increase confidence in the results.

Utilizing containerized tools with RPC. Nugget interacts with these service containers via remote procedure calls (RPC). Upon receipt of an RPC connection, a container executes its particular set of forensic tools on the given data input. In the current implementation, this means that *nugget* uploads the data to a *Docker* container via an RPC function; the container caches the data locally, and subsequent RPC calls issued by *nugget* operate on the cache.

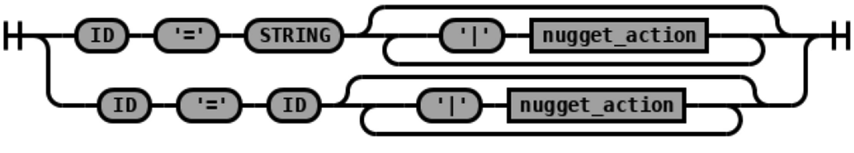


Fig. 3. A railroad diagram of assignment clauses.

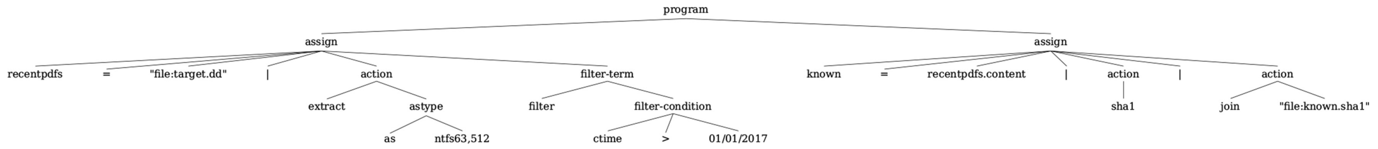


Fig. 4. Sample AST generated by *nugget*.

A sample protocol diagram is shown in Fig. 5, and represents initial forensic steps when investigating an NTFS image — namely, retrieving a listing of all files using *TSK*'s *fls* tool. *Nugget* parses the response, storing resulting data structures in memory.

Utilization of container-based RPC has at least two significant advantages. First, it is readily *extensible*: new commands can be integrated into containers by defining a function that conforms to a single standard, and adding a reference to it in *Nugget*'s source code. Second, it allows for *scaling* of the forensic operation. As shown in (Stelly and Roussev, 2017), networked containers can be configured to distribute expensive forensic tasks, yielding a near-linear increase in throughput-per-container for many typical forensic tasks.

Dynamically extending nugget

Nugget provides a mechanism that allows for it to be extended itself, providing a way for developers to easily add extractors, filters, transformers, and serializers into the base language.

The process for extending *nugget* with new functionality is: 1) identify the type(s) of data which the function will consume and produce, 2) incorporate the new functionality into a container (using a provided template), and 3) run a provided build tool *build-nugget*. *build-nugget* generates and inserts into the defined *nugget* grammar appropriate terminal nodes corresponding to the intended functionality. Further, it generates template code for accessing the *Docker* container via RPC, allowing even novice developers to extend functionality.

The build tool, executed at the user's discretion, will look in a subdirectory and parse all *json* files — one for each transform. A sample is given in 3, which illustrates how *sha1* is added to the grammar.

Listing 3. Extending *Nugget*-sha1.json.

```
{ "name": "sha1",
  "consumes": ["bytes"],
  "produces":  "strings",
  "RPCPort": 2000 }
```

Test case: M57

To test the viability of *nugget*, we have used it to perform a token digital investigation using the realistic M57 patents scenario dataset (Woods et al., 2011). What follows are walkthroughs of three common scenarios: 1) analysis of a hard disk, 2) analysis of a network capture, and 3) analysis of a memory capture. In our prototype implementation, we have incorporated three common tools: *The Sleuth Kit*, *Volatility*, and *tshark*.

As per the original description (Garfinkel): "The 2009-M57-Patents scenario tracks the first four weeks of corporate history of the M57 Patents company. The company started operation on Friday, November 13th, 2009, and ceased operation on Saturday, December 12, 2009."

The data includes daily snapshots of the hard disks and memory of four *Windows* computers, a (nearly) complete network capture, and images of USB devices.

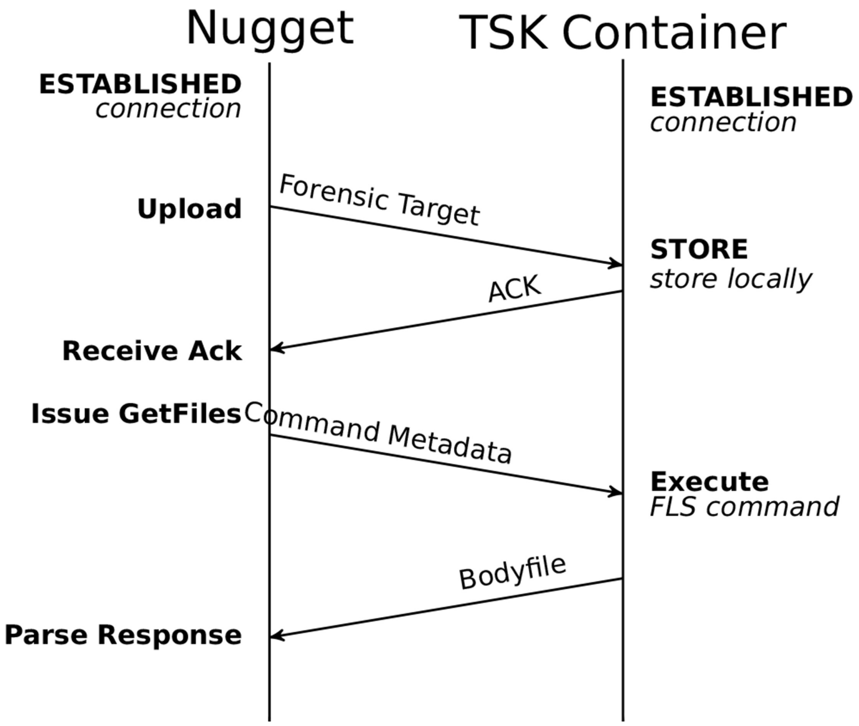


Fig. 5. Sample protocol for retrieving file metadata.

Storage analysis: Integrating TSK

Analysis of hard drive partitions is performed via integration with *TSK* (Carrier). To understand this process, consider the sample *nugget* code in Listing 4. The goal is to read a local file (disk image), extract it as an NTFS image, filter files, perform some hashing, and compare it to a list of known (bad) hashes.

Our first line of code establishes a new variable *files*, which references a local file named `jo-1124.raw`. We then must instruct nugget how to consume this file – that is, we explicitly state that this is an NTFS partition starting at byte offset 63 with a sector size of 512 bytes. The results of this extraction is the metadata for all files within the partition, and is obtained with tools from TSK - specifically, the RPC command instructs the container to run fls on the uploaded image file and return the resulting bodyfile, which nugget then parses.

As previously discussed, this will occur via an RPC to a TSK container (Fig. 5). In this case, the RPC is configured when *nugget* parses the syntax "extract as *ntype*", where *ntype* can be one of a variety of supported types. Further parameters for the RPC are established using other elements of the AST - namely, the byte offset and sector size parameters.

The next line of code requires no external tool call, but rather establishes a local set of filters which will be applied to any preceding actions (recall that *nugget* utilizes lazy evaluation). In this case, our filter will iterate through the results of the *files* variable and yield those files whose name ends with the *jpg* extension.

Next, the *jpghashes* variable will iterate through the results of the *jpgs* variable, and establish the configuration of an RPC call to a SHA1 container due to the *sha1* statement.

This is a good opportunity to show how the *sha1* statement is integrated into the language using the provided build tool, *build-nugget*. Specifically, when *build-nugget* runs, it first reads a `sha1.json` specification file and inserts the specified keyword sha1 into the ANTLR grammar. Next, it generates a `sha1.go` file containing a sha1 action type, which conforms to an interface definition; that is, several specific functions are exposed.

All transforms within *nugget* are coded to an *interface*. Programming to an interface allows *nugget* code to call generic base functions. For example, one of the exposed functions is *GetResults* which executes an RPC against the designated Docker container. Because all transformers *must* expose a *GetResults* method, *nugget* can reliably call it on any transform – md5, sha1, TSKGetFile, etc. Notably, this is all generated for the user based on a few simple lines of JSON. The only task remaining (to the user) is to build a function-specific container which has a forensic tool installed. Once the container is built, the user runs generated code to expose an RPC method, specifying only how to run the forensic tool on the data provided via RPC.

The penultimate line of code establishes a `join` operation, whereby a newline delimited file is compared to the results of the jpghashes variable. Our final line presents results to the user with a `print` operation 5. It relies upon the results of matched, which in turn relies upon the results of jpghashes, etc., causing all dependencies to resolve.

Listing 4. Join operation to find known files.

```
1  files = "file:jo-1124.raw" |
2       extract as ntfs [63,512]
3  jpgs = files | filter name==".*JPG"
4  jpghashes = jpgs.content | sha1
5  matched = jpghashes | join file:kitty.sha1
6  print matched
```

Listing 5. Nugget NTFS Analysis Results.

```
.../Jo/.../hr_patent19.JPG       3a42793...
.../Jo/.../hr_patent20.JPG       34ad6b8...
.../Jo/.../hr_patent21.JPG       17329c9...
.../Jo/.../hr_patent22.JPG        426fe7d...
... [80 further results omitted] ...
```

Network analysis: Integrating tshark

Another critical component of forensic work is investigating network traffic sent and received by a suspect network. Here, network analysis is accomplished with the use of *tshark*. We will look at an example searching for suspicious HTTP GET request, referencing the *nugget* code in listing 6.

The ANLTR-generated parser builds an AST for the input, allowing *nugget* to walk the tree's nodes and execute corresponding functions along the way. For example, the 'filter' phrase in lines 2–3 cause a function named 'EnterFilter' to be called when it is encountered during the walk. Within the function, there are exposed methods to access terminal nodes, such as the string 'tcp and dst port 80 and http', allowing *nugget* to parse the input and setup an internal representation of the indicated filter. In this case, the filter is using the Berkeley Packet Filter syntax. It is applied to its preceding operation (an extraction) when the results of the preceding operation are required. The result will be a collection of all packets which match the filter, and stored in the variable *http*.

Functions exist for every type of node possible in the AST. Lines 4 and 7 are represented in the grammar as a *SingletonOperation* as it is the name of the EBNF production which matches the input. As such, the resulting function call is *EnterSingletonOperation*, with access methods for the term 'http' and 'gets'. Within this function, *nugget* first obtains the evaluation of the indicated variable and prints results using the type's specific print routine.

Listing 6. Nugget and HTTP.

```
1  packets = "file:nov-19.pcap" | extract as pcap
2  http = packets | filter
3      packetfilter=="tcp and dst port 80 and http"
4  print http
5  gets = http | filter
6      packetfilter=="http.request.method=='GET'"
7  print gets
```

Listing 7. GET Requests.

```
patft.uspto.gov   /netacgi/...time+machine
patft.uspto.gov      /netacgi/...immortality
www.google.com  /search?q=steganography...
... [5560 further results omitted] ...
```

Memory analysis: Integrating volatility

In our prototype implementation, memory analysis is performed by *Volatility* (Ligh et al., 2014). To examine this more closely, we will obtain the list of running processes by issuing a *pslist* command - a common initial step when inspecting memory.

In the first line of listing 8, we establish an *extraction* operation on a memory dump. When the AST walk encounters such a node, an internal representation of the extraction is configured and cached until its evaluation is necessary.

On line 3 of the input, we indicate that the results of the extraction (memory) should be given to a *pslist* operation. Internally, this consists of creating an object to store information about the operation. In this case, it needs to track that its input will be the *memory* variable. As this object represents a transform and is implemented as an interface, the object exposes a *GetResults* function. This function is of particular importance to the lazy evaluation process as subsequent evaluations will rely on calling it.

When the final line of code is executed, the variable *procs* is retrieved. Because the variable has yet to be evaluated (an attribute tracked on every transform and extraction), *procs*' *GetResults* function is evaluated. In turn, the variable *memory* is evaluated, and the *GetResults* process repeats.

Memory's *GetResults* function, having previously been configured as an extraction operation for memory, is finally executed. In the case of memory extractions, this involves uploading the specified file to a Docker container with *Volatility* installed via RPC. This function returns a simple acknowledgment to its caller - in this case, from the evaluation of *procs*. As a *pslist* operation, it is configured to make an RPC to the same container. Specifically, the executed command runs a *Volatility* operation to return the list of running processes. This list of processes is then consumed by *nugget* into an internal representation, allowing for the *print* command to access the subfields *name* and *pid*.

Listing 8. Nugget Memory Analysis.

```
1 memory = "file:pat-1203.ram" |
2        extract as memory
3  procs = memory | pslist
4  print procs.name procs.pid
```

Listing 9. Nugget Memory Analysis Results.

```
System                  4
smss.exe              828
csrss.exe             924
winlogon.exe          948
services.exe          992
lsass.exe            1004
svchost.exe          1168
ToolKeylogger.exe    2360
... [24 further results omitted] ...
```

Conclusion

In this work, we presented the concept and initial prototype implementation of the first domain-specific language (DSL), called *nugget*, aimed at providing a practical *and* formal description of digital forensic investigations as a computation. *Nugget* uses ideas from data flow and functional languages to provide high-level domain abstractions that allow forensic analysts to *fluently* and succinctly express the forensic process in a step-by-step fashion.

Nugget provides a declarative, tool-independent means of specifying the necessary processing steps of a forensic case from source evidence to final results. This allows for multiple competing implementations to be employed and compared. Our prototype implementation integrates the language with *Docker*'s cluster runtime to enable parallel execution.

Importantly, *nugget* is designed as an extensible platform for tool integration; new functions can be added seamlessly, and the language can continue to grow to match the needs of analysts.

The widespread adoption of *nugget* would provide a multitude of benefits long sought in the forensics community. Specifically, it would: a) greatly facilitate tool testing and validation; b) enable seamless cross-tool integration (among tools supporting it); c) provide a common language for educational and training purposes; and d) open up the field to the use of big data and AI methods, and high-performance (cloud) services.

Future work. *Nugget* is still early in its development and there is a long list of needed improvements on our agenda. The short list includes IDE support (syntax highlighting and code completion), integration of more functions out of the box, query optimizations, and enhancements to the cluster execution runtime.

References

Apache Hadoop. http://hadoop.apache.org/.

Apache Pig. http://pig.apache.org/.

Bos, J.V.D., Storm, T.V.D., 2011. Bringing domain-specific languages to digital forensics. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 671–680. https://doi.org/10.1145/1985793.1985887. ICSE.

Carrier, B.The Sleuthkit. http://www.sleuthkit.org/sleuthkit/.

Carrier, B., Spafford, E., 2006. Categories of digital investigation analysis techniques based on the computer history model. In: Proceedings of the 2005 Digital Forensic Research Conference (DFRWS), pp. S121–S130. https://doi.org/10.1016/j.diin.2006.06.011.

18 US Code § 1030-Fraud and related activity in connection with computers. https://www.law.cornell.edu/uscode/text/18/1030.

Cohen, M., Garfinkel, S., Schatz, B., 2009. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. In: Proceedings of the Ninth Annual Digital Forensic Research Conference (DFRWS), pp. S57–S68. https://doi.org/10.1016/j.diin.2009.06.010.

Comprehensive Crime Control Act of 1984. https://www.ncjrs.gov/App/publications/Abstract.aspx?id=123365.

Fowler, M., 2010. Domain-specific Languages. Addison Wesley. ISBN: 978–0321712943.

Garfinkel, S. Digital Corpora: M57-Patents Scenario. http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario.

Garfinkel, S., 2010. Digital forensics research: the next 10 years. In: Proceedings of the 2010 DFRWS Conference. https://doi.org/10.1016/j.diin.2010.05.009.

Garfinkel, S., Malan, D., Dubec, K.-A., Stevens, C., Pham, C., 2006. Advanced Forensic Format: an Open Extensible Format for Disk Imaging. Springer New York, Boston, MA, pp. 13–27. https://doi.org/10.1007/0-387-36891-4_2.

Garfinkel, S., Nelson, A.J., Young, J., 2012. A general strategy for differential forensic analysis. In: 12th Annual Digital Forensics Research Conference, pp. S50–S59. https://doi.org/10.1016/j.diin.2012.05.003. DFRWS'12.

Ghosh, D., 2010. DSLs in Action, first ed. Manning Publications Co. ISBN: 978–1935182450.

Gladyshev, P., Patel, A., 2004. Finite state machine approach to digital event reconstruction. Digit. Invest. 1 (2), 130–149. https://doi.org/10.1016/j.diin.2004.03.001.

Ligh, M.H., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory, first ed. Wiley. ISBN: 978–1118825099.

Mernik, M., Heering, J., Sloane, A.M., 2005. When and how to develop domain-specific languages. ACM Comput. Surv. 37 (4), 316–344. https://doi.org/10.1145/1118890.1118892.

Odersky, M., Spoon, L., Venners, B., 2016. Programming in Scala: Updated for Scala 2.12, third ed. Artima Press. ISBN: 978–0981531687.

Parr, T., 2013. The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, second ed. ISBN: 978–1934356999.

Pike, R., Dorward, S., Griesemer, R., Quinlan, S., 2015. Interpreting the data: parallel analysis with Sawzall. Sci. Program. 13, 277–298. https://doi.org/10.1155/2005/962135.

Pirolli, P., Card, S., 2005. Sensemaking processes of intelligence analysts and possible leverage points as identified through cognitive task analysis. In: Proceedings of the 2005 International Conference on Intelligence Analysis. http://researchgate.net/publication/215439203.

Puppet Labs, 2015. Pup-987 Remove Ruby DSL Support. https://tickets.puppetlabs.com/browse/PUP-987.

Roussev, V., 2015. Building a forensic computing language. In: 48th Hawaii International Conference on System Sciences. https://doi.org/10.1109/HICSS.2015.617.

Roussev, V., 2016. Digital Forensic Science: Issues, Methods, and Challenges, first ed. Morgan & Claypool Publishers. ISBN: 978–1627059596.

Roussev, V., Ahmed, I., Barreto, A., McCulley, S., Shanmughan, V., 2016. Cloud forensics – tool development studies and future outlook. Digit. Invest. 18 (Suppl. C), 79–95. https://doi.org/10.1016/j.diin.2016.05.001.

Scientific Working Group on Digital Evidence (SWGDE). SWGDE Documents. https://www.swgde.org/documents.

Scowen, R., 1998. Extended BNF – a generic base standard. Tech. rep. Technical Report, ISO/IEC 14977 http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

Stelly, C., Roussev, V., 2017. SCARF: a container-based approach to cloud-scale digital forensic processing. In: 17th Annual Digital Forensic Conference (DFRWS), pp. S39–S47. https://doi.org/10.1016/j.diin.2017.06.008.

SweetScape Software Inc. 010 Editor - binary templates. https://www.sweetscape.com/010editor/templates.html.

Wirth, N., Nov 1977. What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM 20 (11), 822–823. https://doi.org/10.1145/359863.359883.

Woods, K., Lee, C., Garfinkel, S., Dittrich, D., Russell, A., Kearton, K., 2011. Creating realistic corpora for security and forensic education. In: Proceedings of the ADFSL Conference on Digital Forensics Security and Law, pp. 123–134. http://commons.erau.edu/cgi/viewcontent.cgi?article=1161&context=adfsl.