Fall 12-20-2019

A Domain Specific Language for Digital Forensics and Incident Response Analysis

Christopher D. Stelly
cdstelly@gmail.com

Recommended Citation

Stelly, Christopher D., "A Domain Specific Language for Digital Forensics and Incident Response Analysis" (2019). *University of New Orleans Theses and Dissertations*. 2706.
https://scholarworks.uno.edu/td/2706

A Domain Specific Language for Digital Forensics and Incident Response Analysis

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science
Computer Science

by
Christopher Drew Stelly
B.S. University of Louisiana 2012
M.S. University of New Orleans 2014
December 2019

Declaration of Authorship

I, Christopher STELLY, declare that this thesis titled, “A Domain Specific Language for Digital Forensics and Incident Response Analysis” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

UNIVERSITY OF NEW ORLEANS

Abstract

College of Science
Department of Computer Science

Doctor of Philosophy

A Domain Specific Language for Digital Forensics and Incident Response Analysis

by Christopher STELLY

One of the longstanding conceptual problems in digital forensics is the dichotomy between the need for verifiable and reproducible forensic investigations, and the lack of practical mechanisms to accomplish them. With nearly four decades of professional digital forensic practice, investigator notes are still the *primary* source of reproducibility information, and much of it is tied to the functions of specific, often proprietary, tools.

The lack of a formal means of specification for digital forensic operations results in three major problems that go to the core of digital forensic science and its practical application. Specifically, there is a critical *lack of*: a) standardized and automated means to scientifically verify accuracy of digital forensic tools; b) automated methods to reliably reproduce forensic computations (their results); and c) common framework for inter-operability among disparate forensic tools. In addition, there is no standardized means for communicating software requirements between users (investigators) and researchers and developers, resulting in a mismatch in expectations, especially with respect to performance. Combined with the exponential growth in data volume and the complexity of applications and systems to be investigated, all of these concerns result in major case backlogs and inherently reduce the reliability of the digital forensic analyses.

This work proposes a new approach to the formal and usable specification of forensic computations, such that the above concerns can be addressed on a sound scientific basis via the introduction of a new *domain specific language* (DSL) called *Nugget*. DSLs are *specialized* languages that aim to address the concerns of particular domains by providing practical abstractions that allow users to directly provide an intuitive, but formal, description of their problem statements. Successful DSLs, such as SQL in relational databases, can transform an application domain by providing a standardized way for users to communicate what they need without specifying how the computation should be performed.

This is the first effort to build a DSL, and a prototype support infrastructure for (digital) forensic computations with the following research goals: 1) provide an intuitive formal specification language that covers core types of forensic computations and common data types; 2) provide a mechanism to *easily* extend the language that can incorporate arbitrary computations; 3) provide a prototype execution environment that allows the fully automatic (and optimized) execution of the specified computation; 4) provide a complete, formal, and auditable log of computations that can be used to independently reproduce the results of an investigation; 5) demonstrate *scalable*, cloud-ready processing that can match the growth in data volumes and complexity with necessary hardware resources.

Keywords. nugget,scarf, digital, forensics, cyber, security

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AFF	**A**danced **F**orensics **F**ile format
API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
BNF	**B**ackus **N**aur **F**orm
DF	**D**igital **F**orensics
DSL	**D**omain **S**pecific **L**anguage
ENISA	**E**uropean **U**nion **A**gency for **C**ybersecurity
ES	**E**lastic **S**earch
FBI	**F**ederal **B**ureau of **I**nvestigation
GNOCIA	**G**reater **N**ew **O**rleans **C**enter for **I**nformation **A**ssurance
GRR	**G**oogle **R**apid **R**esponse
GUI	**G**raphical **U**ser **I**nterface
HDD	**H**ard **D**isk **D**rive
IDE	**I**ntegrated **D**evelopment **E**nviornment
IO	**I**nput **O**output
IOC	**I**ndicator **O**f **C**ompromise
JSON	**J**ava **S**cript **O**bject **N**otation
NFI	**N**etherlands **F**orensics **I**nstitute
NIST	**N**ational **I**nstitue of **S**tandards and **T**echnology
NSF	**N**ational **S**cience **F**oundation
RCFL	**R**egional **C**omputer **F**orensic **L**abratory
RPC	**R**emote **P**rocedure **C**all
SCARF	**SCA**lable **R**ealtime **F**orensics
SWGDE	**S**cientific **W**orking **G**roup on **D**igital **E**vidence
SSD	**S**olid **S**tate **D**rive
SQL	**S**tructured **Q**uery **L**anguage

For my family.

Chapter 1

Introduction

Digital forensic science, often referred to as *digital forensics*, is the use of scientific methods to process digital artifacts in order to provide evidence in support of legal proceedings. Fundamental to the credibility of this mission is the requirement of *reproducibility*, which has been handled (primarily) in an ad-hoc manner. Independent reproducibility of experimental results is a central requirement for any valid scientific work – it should be possible for anyone (with suitable qualifications and experimental equipment) to reproduce published scientific results based on the description provided by the authors. In a *forensic* context, reproducibility is the central to the justification for using scientifically-derived evidence in any legal proceedings.

Digital forensics (DF) has a particularly thorny version of the problem in that the fast pace of information technology creates an enormous, and fast growing, amount of data, which is processed in an ever-increasing number of ways by applications. As a consequence, new forensic tools need to be developed, updated and integrated at the cost of ever increasing complexity, making the goal of (manually) *validating* the reproducibility of the results ever more infeasible. The root issue is that there has been no standard specification for DF computations that is both *practical* (it can express actual forensic tasks) and *formal* (it can be mapped to running code). This causes many problems, but the most important consequence is that practical verification of digital forensic can only performed on relatively few (basic) types of computations.

Our research addresses these issues with the design and implementation of a domain specific language (DSL) built specifically for DF. Our language establishes a layer of abstract, yet formal, communication between investigators, tool developers, and the legal system. Our solution is at the core layer, providing a robust foundation for solutions to problems across the *entire domain* of digital forensics — from educational materials to tool verification. In short, we aim to do for digital forensics what SQL did for the relational database.

First, though, we must examine what digital forensics is, where it originated, and how we have arrived at its current state. We then detail issues facing the domain. Finally, we discuss the background information necessary to understand both related work and our solution, which are detailed in Chapters 2 and 3, respectively.

1.1 Definitions, Usage, and Models

There are two commonly used definitions of digital forensics. The first Digital Forensics Research Workshop (*DFRWS*), organized in 2001, provided one of the most frequently cited definitions of digital forensic science in literature:

> Digital forensics is the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitation or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations. [28]

' This definition emphasizes event reconstruction to identify and react to threats in a security incident. By contrast, the definition offered by the National Institute for Standards and Technology (NIST) is:

> digital forensics is considered the application of science to the identification, collection, examination, and analysis of data while preserving the integrity of the information and maintaining a strict chain of custody for the data. Data refers to distinct pieces of digital information that have been formatted in a specific way. [47]

This definition emphasizes a legal perspective, which requires the flexibility often necessary during legal proceedings.

These definitions are generally useful, but do not provide a technical starting point for our research. For this, we employ a version of the working definition offered by Roussev [23], which seeks to connect :

> Digital forensics is the process of identifying and reconstructing the relevant sequence of events that have led to the currently observable state of a target IT system or (digital) artifacts.

This definition contains many notable aspects. First, if we were to remove the references to both "digital" and "IT systems" from this definition, we would have a definition readily recognizable as the approach for traditional law enforcement forensic analysis. Second, we have references to tools, methods, and artifacts, which merit individual discussions necessary to understand this research.

1.1.1 Methods, artifacts, tools, and targets

Targets. Traditional targets of DF include consumer or industrial products, personal computers and laptops, as well as corporate servers. Non-traditional targets would include digital platforms such as smartphones and internet-of-things-enabled devices; however, as societal adoption drives these platforms into the mainstream, we can expect them to become increasingly prevalent targets of digital investigations. During such investigations, the investigator must identify, collect, and analyze specific pieces of data (or artifacts) from targets.

Artifacts. Most digital forensic artifacts fall within one of three broad categories, based on their acquisition. The categories are **file system**, **network**, and **memory**.

File system artifacts are generally the most common target of investigations. At a technical level, this simply implies that the artifact resides on a system disk (such as a hard drive, solid-state drive, or USB thumb drive). An investigator might be interested in numerous types of files, such as illegal pictures, incriminating emails saved to disk, or a piece of executable malware – the list is interminable. However, there are also other aspects to file system artifacts.

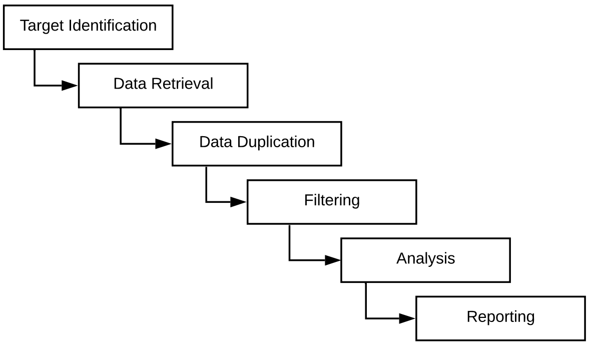


FIGURE 1.1: Generic Forensic Methodology

In particular, their metadata (such as file creation time or file hash) is of vital importance to the investigator.

Network artifacts are more complicated to collect and analyze. While it is common for corporations and governments to require full network logging (that is, saving a copy of all network traffic for potential analysis), this logging is not necessary for the average computer user. Network artifacts are thus less accessible simply because they may not be available, at least depending on the investigation. In cases where they are available, investigators face two further difficulties: first, they must reconstruct network artifacts from the original network logs (including filtering suspect traffic from legitimate traffic), and second, network traffic is increasingly becoming encrypted, thereby preventing any investigation into the content of the traffic (although the analysis of metadata is a potential option).

Memory artifacts are generally the most difficult for the investigator to collect, but are also (usually) the easiest to analyze. These artifacts exist in RAM, or volatile (temporary) storage, which is permanently lost after a computer shuts down or reboots. However, when investigators have the "memory dump" of a target system available, they are granted a direct view of the state of the system. Some examples of what can be extracted include a list of running processes (with in-memory artifacts), evidence of malicious process injections, and passwords. Analysis of retrieved memory artifacts is generally less difficult than analysis on other types of artifacts because they are usually unencrypted, giving the analyst a *direct* "snapshot view" of the IT system at the point in time of the memory capture.

Methods. While numerous separate digital forensic methodologies exist, they broadly follow the same steps to obtain artifacts from their forensic targets. As illustrated in Figure 1.1, investigators must first identify and retrieve their target, obtain the target's data, safeguard (or duplicate) the data, and perform filtering; only then can they begin an analysis with their preferred tools and procedures [82, 65, 73].

Target identification and retrieval are largely dependent on the investigation. During legal proceedings, law enforcement analysts are generally restricted to retrieving targets identified in a warrant, and can be further restricted by privacy restrictions for privileged information (such as doctor-patient confidentiality). In corporate settings, such as during an internal audit, investigators generally have authority to execute investigations on any corporate-owned device. Once a target is within the scope of an investigation, analysts can obtain the data storage device.

Obtaining data from the target is fairly simple in most cases: physical acquisition of the target hard drive, USB stick, or memory capture is the norm. Numerous types of physical devices allow for the connection of acquired devices to forensic workstations. In more complicated cases, the investigator may have to counter anti-forensic precautions taken by a malicious actor. That is, advanced criminals may have built-in automatic protections from forensic investigations, and the investigator must take these into account during this phase.

Duplication of data is a vital step in the forensics process. The primary goal is to preserve target data for consequential legal proceedings. This usually involves a purpose-built device, which can copy data to a read-only medium from which investigators will conduct their investigation.

Filtering data, akin to *triage*[44, 79], is an important step for efficiency. It usually consists of identifying potentially relevant artifacts while simultaneously ignoring known innocent artifacts. For example, in a case of insider trading, investigators would be interested in artifacts related to websites visited, emails sent, etc., while simultaneously ignoring one's music library. This type of approach is necessary because of the large volume of data present during investigations, as tools which investigators utilize are generally more efficient at smaller scales.

Tools and analysis. Once the data is safely acquired, the investigator can begin his analysis. Analysts utilize a variety of digital tools, which are necessary because of both sophisticated underlying technologies and the sheer scope involved in a DF investigation.

Tools is a generic term encompassing everything from simple programs written by individuals to large-scale, vendor-supplied products. These tools are generally responsible for performing operations on data (artifacts); more specifically, they are responsible for everything from finding artifacts (filtering) to reporting on what those artifacts contain. As an example, the most popular free DF toolkit is "The Sleuth Kit"[1], which allows investigators to, among other things, search for, filter, and extract files from hard drives. On the vendor-supplied front, "EnCase"[2] is the *de facto* standard used across both law enforcement and corporate environments. A plethora of other tools fulfills smaller purposes, such as establishing a timeline of events or simply cataloging data for the investigator.

Now that we have established a context of what the term "digital forensics" encompasses, we turn our attention to *who* uses these tools and methods.

1.1.2 Digital forensics usage

The traditional end-user of DF is law enforcement. However, both corporations and researchers are heavy practitioners of DF tools and practices.

[1] https://www.sleuthkit.org/

[2] https://www.guidancesoftware.com/

Law Enforcement. From international courts to local courts, all levels of law enforcement are becoming familiar with DF. As computerized technology increasingly moves into the mainstream, law enforcement's role in DF will continue to grow. Indeed, if recent F.B.I. case backlogs are any indication of broader trends [100], then law enforcement everywhere will be forced to dedicate significant resources to DF.

Typical law enforcement analysts are tasked with proving a prosecutor's charge. This could involve disproving an alibi or finding evidence of child exploitation on a hard drive. Depending on the specific task, they will utilize established methodologies and tool sets while simultaneously noting their conclusions for legal analysis. Their notes are often the main source of evidence for judicial proceedings.

Corporations. Corporations generally have access to their own devices. If a corporation suspects an employee of malicious behavior, it may initiate an investigation into the employee's computer. Corporate investigations do not require a warrant; however, they often require the approval of a human resources department. Once approved, the analyst working the investigation will utilize the same previously outlined tools, techniques, and methodologies.

On the other hand, DF techniques are employed when malware (such as viruses or trojans) are suspected on a corporate network. Analysts work to quickly identify the source of infection and study its effects in order to deploy appropriate countermeasures. Similar tools, techniques, and methodologies are employed.

Researchers. Finally, DF is an extensive research topic. Not only are researchers contributors of open-source digital forensic tools, but they can also provide beneficial services. A common case is data recovery – that is, retrieval and recovery of accidentally deleted data.

Next, we discuss how investigators, analysts, and researchers have brought the field of DF to its current state.

1.2 Brief History of Digital Forensics

The first official steps towards the professionalization of digital forensic investigations in the US date back to the 1980s, and stem from the passage of the first legislative acts on computer crime (the Comprehensive Crime Control Act) [15, 1], and the establishment of F.B.I.'s *Magnetic Media Program* in 1984 [30]. 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.

During the 1990s and early 2000s, DF matured into a more professional realm. Until this point, many tools were produced by hobbyist practitioners; however, researchers began to release robust tools, many of which are still in use today. Vendors entered the market with full-featured solutions for sale. DF also became the subject of academic research across the world, with several institutions publishing works on the subject. The mid-2000s saw the establishment of various government treaties on cyber-crime laws, as offered by the *Convention on Cybercrime* in 2004.

Since then, the field of DF has steadily advanced. From policy and legal precedents to research and new tools, the field is evolving rapidly. As we detail later in this chapter, this evolution is occurring too rapidly. Briefly, little science has been injected into the DF process. That is, during the organic growth of the field since the 1980s until today, there has been no significant standardization or unification of the various tools and procedures. The end result is an unstable state: investigators have hundreds of tools to choose from, dozens of methodologies they can follow, and an ever-changing technological landscape. Before detailing these arguments, however, we discuss a few notable real-life investigations and how DF played an important role.

1.2.1 Examples and notable cases

BTK, 1991. One infamous case of early DF involved a serial murderer known as **BTK**. Between 1974 and 1991, BTK, who was later identified as Dennis Rader, killed 10 people. However, he was not immediately caught (or even identified). By 2004, it was considered a cold case when Rader began taunting law enforcement with various forms of communication to police departments and news stations. In early 2005, he issued his first digital communique: a Microsoft Word document stored on a 1.44-MB floppy disk that he physically mailed to a news affiliate. While the *content* of the note did not identify Rader, law enforcement, using the various techniques and procedures described above, was able to retrieve *digital metadata* from the document (known colloquially as an *artifact*). The metadata included a reference to his church and was directly responsible for Rader's subsequent capture and arrest. He is now serving multiple life sentences.

While the previous example has showcased how DF plays an important role in helping law-enforcement investigate traditional "non-cyber" crimes, it clearly plays a critical role of investigations into "cyber-crimes". As given by Merriam-Webster:

> Cybercrime is criminal activity (such as fraud, theft, or distribution of child pornography) committed using a computer especially to illegally access, transmit, or manipulate data[59]

In other words, crimes that are executed primarily with the use of digital assets, such as computers and mobile devices.

James M. Cameron, 2009. An investigation into James M. Cameron exemplifies a prototypical DF case. In early 2009, Cameron (then an assistant attorney general for the state of Maine), was indicted on 16 charges of trafficking child pornography for possessing illicit imagery hosted on his Yahoo photo album. Interestingly, the investigation began when Yahoo contacted authorities — not the other way around (indeed, Cameron's rejected defense hinged on this fact). Investigators from the Main State Police Computer Crimes Unit retrieved four computers from his residence, and using the techniques and methods discussed above were able to retrieve incriminating chat transcripts which ultimately identified Cameron as the perpetrator.

Morris Worm, 1988. In November 1988, a graduate researcher named Robert Morris released a piece of malware which became known as the Morris Worm. It exploited weakness in various programs, such as *finger*, *sendmail*, and *rsh*, eventually causing a denial of service condition to infected machines [26]. Although not intentionally malicious, Morris was eventually found guilty after the FBI located him and analyzed his computer for evidence [27]. Morris was the first to be found guilty of the Computer Fraud and Abuse Act of 1986 [1]; later analysis from the Government Accountability Office's estimated damages of between $\$100,000$ and $\$10,000,000$ [93]. The important note is that the F.B.I. was able to use (relatively) nascent DF processes to prove guilt. [3]

Now that we have a firm understanding of what digital forensics is, an idea of how we got here, and a few examples of its usage, we can detail broad problems facing the field.

1.3 The Problems within Digital Forensics

With the advancement and widespread adoption of digital technology, the need for tools and processes to assist investigators in investigations became a primary concern of law enforcement and researchers alike. Many tools of varying design were thus designed and built. Tools were often purpose-built for specific problems, such as extraction of file metadata from a Windows file system.

However, it is vitally important to emphasize that these tools were built to answer specific, *contemporary* needs. Compared to today, tools were developed when hard drives were orders of magnitude smaller, data transfer rates were orders of magnitude slower, and processors were orders of magnitude slower. For comparison, supercomputers in the 1980s executed less floating operations per second (FLOPS - a common performance metric) than modern smartphones. To complicate matters, changes to underlying technology (such as file system architecture) occur at a relatively rapid pace. In short, tools built by researchers tended to solve a specific problem, for a specific technology, at a specific point in time; at no point did tool builders comprehensively address the "big picture".

The situation can be succinctly summarized as follows: *the field of digital forensics has yet to produce a holistic approach towards a practical and computationally formal description of forensic operations*.

The lack of such a scientific description directly contributes to a myriad of issues across the field of DF. First, the results of DF investigations are difficult to reproduce and/or verify, which is a necessary tenet in any scientific endeavor, and especially pertinent when considering the importance of legal proceedings. Second, investigators are forced to use individual tools (and in many cases, they must have significant programming experience to use those tools effectively); conversely, tool creators often work with limited design requirements. Third, proprietary tools are difficult to verify or benchmark without an accepted standard against which to compare them. Fourth, students currently learn about DF with the use of tools, as opposed to learning the computation (conversely, knowledgeable experts are forced to learn specific tool syntax to execute a desired forensic computation). Fifth, no standard means exist for security tools to interchange data – they are (for the most part) incompatible with one another. Finally, current practices do not scale at a rate commensurate with the amount of storage involved in digital investigations.

We discuss each of these points and then describe our approach to solving them. We have two complementary approaches: developing a domain-specific language (DSL) for DF and developing a scalable runtime framework for it to execute queries against. Our research finds

[3] As a sidenote, the Morris Worm has been popularized in several cultural adaptations [86, 54] and has been credited with creating some of the first demands for cybersecurity response teams [27].

that inserting a layer of abstraction between tools and investigators, where our abstraction is a DSL with scalable runtimes, solves these (and other) systemic issues facing the field of DF.

1.3.1 Replication and Verification

Without a means to describe a DF computation (or series of computations), there is no *standard* way for an investigator to capture the specific steps he took in an investigation. As such, investigator notes are a primary means of cataloging an investigation and are a primary source in law enforcement or judicial proceedings. Having to follow investigator notes is a far cry from being reproducible and is ultimately unscientific (as the ability to reproduce a process, methodology, or study is a key tenet of any scientific endeavor).

During courtroom proceedings, the investigator's work must hold against counter-arguments. This *verification* of the investigator's work is highly subjective when he or she only has notes to reference. Ideally, this would be an objective measurement of his or her steps in the investigative process, which is possible with the adoption of a standardized description of the forensic computation. This allows independent parties to verify that a given forensic computation matches the expected output.

Many attempts have been made to formally describe the DF process, ranging from the abstract to the concrete. We briefly consider each in Chapter 2, but ultimately find that none provide an abstraction usable to the investigator while simultaneously providing *unambiguous computational descriptions* usable as language constructs.

As a direct example, consider the difference in the following two ways to describe matching files to known hashes (a common forensic task). Scenario A represents investigator notes, while scenario B represents usage of a standard computational description:

1. Scenario A) First, launch the Sleuth Kit program and use its file navigation graphical user interface (GUI) to search for all PDFs from the forensic target named "target.dd", which is a Windows-based image with a sector size of 512 bytes and an offset of 63 bytes. Then, calculate the hashes for the selected PDFs using the SHA1 algorithm by clicking on the button in the upper middle of the screen. Then, for each file hash, compare to the list of known hashes in the file "hashes.txt".

2. Scenario B)

   ```
       pdfs = select *.pdf from "file:target.dd" as ntfs
   [63,512]
       hashes = pdfs | hash sha1
       known = hashes | join "file:hashes.txt"
   ```

Both scenarios represent a common digital forensic process - hashing files and comparing the result to known hashes. However, scenario A is a "human-readable" description, such as found in investigator notes, while scenario B is both a computational description *and* a human-readable description, **especially** for forensic analysts. As opposed to the former, which could be written in a myriad of word substitutions and rearrangements, the latter is *precise*, *repeatable*, and *unambiguous*. These traits embody the core tenets behind **domain specific languages**, which we propose as a solution to aforementioned issues.

Domain Specific Languages

As given by Fowler [29], 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:

1. *Formality*. A DSL is a formal language, with established grammar and semantics, that is translated into *executable* code.
2. *Fluency*. An ideal 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.
3. *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*.
4. *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 summary, a DSL grants domain experts the constructs necessary to describe a problem or solution succinctly and efficiently, and it 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 [37].

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 higher levels of automation as seen in large-scale (cloud) environments. Additionally, DSLs are driven by the need of state and state transition formalization 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 or virtual machine (VM) configuration languages by *Docker* and *Vagrant*.

1.3.2 Tools

A large portion of the investigative process involves the use of digital tools. These tools range from free and open-source scripts to robust vendor-supplied solutions. Investigators generally use several tools throughout the course of an investigation. A large tool chest allows an analyst to use specialized tools for specific tasks; however, this approach has drawbacks.

Learning tools. First, learning individual tools imposes a significant learning penalty on the operator. Each tool has its nuances and idiosyncrasies, which operators must learn, if not master, for effective use. Furthermore, tools often change as bugs are fixed, features are added, or legacy support is removed. This volatile 'tool environment' is difficult to manage, *especially* as analysts tend to employ a number of tools.

Specialized tools. Second, utilization of most open-source tools requires a strong programming background. Open-source, often unsupported tools are commonly built to suit a highly specific task and are driven only by community involvement. Other tools never considered volume – that is, they may work well with small targets, but in the face of high-volume input, they will perform poorly. With ever-increasing workloads, this forces the analyst to choose some combination of the following: 1) buy more hardware, 2) write batching code, 3) modify the tool, and 4) find a new tool.

Specifying tool functions. Third, a disparity exists between the investigators who use the tools and the programmers who provide them. Even supposing that a feedback channel exists for requirements between analysts and developers, there is no language whereby investigators can express the *exact* computation they need to have resolved. This ambiguity leads to an imperfect description of needs and results in an imperfect tool. We submit this is driving cause for the findings by Hibshi et al., who discovered significant usability issues of DF tools [43].

Tool interoperability. Digital tools are not currently interoperable. Since they are often specialized and almost always developed independently, they are unable to share data [10]. For a comprehensive forensic solution, any integration between tools is therefore expensive. Although efforts are underway to unify data representation, this would only work on tools that are newly developed or (expensively) retrofitted for conformity.

1.3.3 Tool veracity

While regulated and accepted standards exist that govern many common forensic computations (hashing, for example), there is no standard means by which an *implementation* of the tool executing the computation can be verified. That is, there is no way of determining whether investigators, lawyers, or analysts can trust their tools. On one hand, in cases where source code is available, code review is a viable — albeit costly — option. On the other hand, in the case of vendor-supplied or for-profit tools ("black box"), source code review is not possible. Instead, a dynamic quantitative analysis can yield a high degree of confidence. However, no standard or accepted method currently exists for testing products against known inputs with standardized outputs. Moreover, tools are *constantly* changing to keep up with improvements to underlying technologies [87]. As argued by Carrier (author of the Slueth Kit), forensic tooling should be held to the highest standard possible [6] because of the serious implications of forensic investigations (e.g., firing, criminal convictions).

Tool veracity is an acknowledged problem across the industry with little to no solution in sight. The barrier to veracity testing is the lack of a means to execute testing. However, it is worth noting that tool *studies* are being explored by institutions such as NIST. Their Computer Forensics Tool Testing Program (*CFTT)* studies individual tools in a manual "feature test" process [62]. Testers generate a report based off the tool's performance against a publicly specified test target, and subsequently posts results to the public [63]. This approach is flawed for at least two reasons. First, the testing process begins with manual review of a tool's documentation, from which all test cases are determined. This is a time-consuming process prone to human error, especially given the rapid rate of change of underlying technologies which can render a tool's review invalid [87]. Second, it is a qualitative process. That is, test cases are determined by humans; test cases are executed by humans; and results are analyzed by a committee. A better approach would be an automated, *quantitative* process.

1.3.4 Educational materials

The lack of educational resources in DF is a well studied issue [11]. We submit that a root cause is the lack of a description for the forensic computation. The result is that instructors have no *standard* way of teaching individual steps within a forensic investigation. For instance, to teach students how to retrieve metadata from a digital image, one instructor might talk about how to use the "exif" tool on the command line. Another might show students how to use "The Sleuth Kit". Another could even use operating-system builtin functions. Students ultimately learn how to use individual tools, rather than focusing on the core concept of retrieving metadata.

1.3.5 Scaling

Due to the rapidly decreasing cost of storage, investigators face an ever-increasing amount of data during investigations. The amount of data directly contributes to the overall length of an investigation, and because investigations have real-world deadlines, the overall throughput *per* analyst is *expected* to increase. This expectation is flawed for a few reasons.

First, older tools (which have become industry standard) were never intended to meet the demands of today's storage volumes. They were designed and implemented within a different, specific context – relatively small datasets and forensic targets within a specific sub-domain (such as memory forensics). Furthermore, they were never designed to scale using distributed computing. Experience demonstrates that retrofitting such a fundamental property is difficult, for example the short-lived effort to extend *The Sleuth Kit* with *Hadoop*'s big data processing capabilities.

Second, the growth of datasets has largely been unrealized because of the limited throughput of traditional "spinning" hard disks (the default choice for most desktops). However, advancements in SSD storage – fast-falling costs, advancing capacity, and bandwidth growth – are quickly reshaping the storage technology environment. SSDs, for example, can currently be expected to reach a transfer rate of at least 400 MB/s. They are also not the final evolution – non-volatile memory express (*NVMe*) drives are becoming increasingly popular and affordable, and perform above 1 GB/s.

Third, the number of devices involved in cases is growing at an alarming rate. For example, most people now have advanced smartphones (which are capable of committing illegal activities); "smart" watches, which could contain digital evidence, are now commonplace; vehicles are networked and are full computer suites in their own right; and USB sticks are inexpensive. As technology becomes affordable and miniaturized, it will become pervasive. As this occurs, investigations will target more devices.

Finally, network advances have made a network-based acquisition of forensic data a strong possibility. 10Gb Ethernet, now commonplace in corporate environments, can be expected to be replaced by 100Gb *Infini-Band* in coming years. Looking ahead, IEEE has approved specifications for even faster Ethernet connections - currently supporting up to 400Gbps with plans for 1Tbps. Thus, we can envision tooling to eventually deliver forensic targets over a network, as given by Roussev [78].

Investigators ultimately face not only a volume but a throughput problem: they are facing more data and obtaining it faster than ever before. Their current generation of forensic tools cannot handle the throughput of this data, and their tools are not designed to scale up – they cannot easily "throw hardware at the problem." As a result, investigators require additional time to complete their average case, thus delaying investigations and adding to an ever-increasing backlog.

1.4 Brief description of research

We address these issues with the development of a practical digital forensics DSL, named *Nugget*. The language provides constructs for usability, reproducibility, verification, and tool interoperability. It gives educators tool-agnostic materials to teach DF methodologies and law-enforcement the means to specify computational needs to tool developers. In short, it will do for DF what SQL did for database technologies.

In order to test *Nugget*, we develop a runtime named *SCARF*. *SCARF* is a scalable container-based tool-agnostic architecture allowing us to tie arbitrary DF tools into a single processing “fabric”. This architecture is deployable across cloud-based or distributed platforms and allows us to add arbitrary amounts of hardware to combat the problem of rapid growth of data volume in digital investigations.

1.5 Contribution Claims

We contribute to the science of digital forensics in the following ways:

1. Provide digital forensic investigators the means to specify a digital forensics computation that is both practical and formal;
2. Provide digital forensic investigators a means for reproducing digital forensic investigations;
3. Provide an abstract layer of communication suitable for use between forensic analysts, law enforcement, and tool developers;
4. Perform a mechanism by which digital forensic tools can be benchmarked;
5. Provide external entities (e.g., NIST) a means to validate digital forensics tools;
6. Provide educators a tool-agnostic medium to teach digital forensics processes;
7. Provide a framework enabling interoperability between digital forensic tools;
8. Create a flexible, scalable, and container-based runtime to demonstrate effective usage of *Nugget* queries.

Remaining organization

Our research is organized into five chapters, an appendix, and a bibliography. Chapter 2 presents a review of the literature regarding the current state of DF, with an emphasis on proposed languages, scalable solutions, and model abstractions. Thereafter, Chapter 3 covers the research design and methodology. The results and analysis are presented in Chapter 4, and finally, Chapter 5 contains the summary, conclusions, and recommendations resulting from the research.

Chapter 2

Literature Review

2.1 Background

Our ultimate goal is to bring science into digital forensics by developing a practical domain specific language. We simultaneously need to develop a scalable high-performance runtime platform to combat contemporary technological storage and processing capabilities. Before developing either solution, we conducted a literature review to ascertain whether these problems have been previously addressed, either fully or partially.

We approached our literature review in two broad phases. First, we assessed other approaches for modeling DF computations or investigations. We found that a few tentative solutions exist for extremely narrow "languages"; however, no solution has taken a holistic approach — most solutions are focused on solving particular problems within the domain, and none promise *practical* science. Second, we focused on investigating how others are solving (or attempting to solve) the problem of scaling to current *volume*. We found that a few tentative solutions exist for combating the amount of data that digital investigators are encountering; however, these solutions are ultimately insufficient.

2.2 Modeling a Standardized Approach

2.2.1 Background

Adoption of a standardized approach is key to the future of DF. Without such a movement, Garfinkel has argued [30] that tools will become increasingly obsolete, data volumes will become insurmountable, and results will be ultimately unreliable — especially in courts of law. He continues:

> Digital Forensics is facing a crisis. Hard-won capabilities are in jeopardy of being diminished or even lost as the result of advances and fundamental changes in the computer industry.

Garfinkel is not the only one to draw such a conclusion. In 2006 a DFRWS working group *Common Digital Evidence Storage Format* was established to address the recognized need for standard evidence formatting [40]. And, for example, Guo et al. directly challenge the current lack of "a validation paradigm" in DF, especially given the need for accurate, reliable, and verifiable results [41]. We discuss both of these works in more detail below.

After performing a thorough literature review, we have divided related works into three broad categories: high level models, low level/mathematical models, and DF languages.

2.2.2 Low-Level and Mathematical Models

In this section, we briefly discuss models that are too low-level. These models are useful in special cases, such as providing theoretical solutions, but they are not *practical* for end users.

Carrier's history testing model

In [9], Carrier et al. describe two general models: *primitive computer history* and *complex computer history*.

Primitive compute history has its roots from finite state machines (FSM). Carrier explores extrapolating FSMs into a representation of every possible state of the computer based on low-level input events. Every unit of storage is treated as a state, such as registry, memory, and hard disk locations. Transitions between state changes are a historical accounting of the computer behavior. He notes that while possible to represent a *theoretical* computer in such a mapping, it is non-trivial or impossible to represent practical computer history in such a manner.

Complex computer history, on the other hand, is a similar model but built to represent *complex events* - those which cause lower level primitive events to occur. For example, saving a file is a single event to the user, but causes hundreds (or thousands) of primitive level events to occur. These events are grouped together as complex events according to pre-defined groupings, which in turn provide a more reasonable representation of a computer's history.

Carrier goes on to describe possible uses for such a low-level model, such as representing various classes of analysis techniques. However, usage of finite state machines to represent computer history is clearly too low-level for any practical use across DF.

Gladyshev's finite state machine

Similar to Carrier, Gladyshev et al. also investigate approaching digital forensics with a finite state machine model [38]. However, this research is different in at least two ways: it is more in-depth, and it focuses on discrete *event reconstruction* using hypotheses - such as, "Did Alice use the printer?". The authors explain an algorithm to automatically explore event reconstruction.

However, FSM-based event reconstruction is computationally expensive, and it is unrealistic for complex events because of the sheer number of state transitions. The authors have acknowledged that further research is required to benefit the everyday investigator. For these reasons, in addition to the FSM-related shortcomings discussed with Carrier's history model, this approach does not solve the issues we have presented.

Garfinkel's differential analysis

Garfinkel et al. explore *differential forensic analysis* [34]. General differential analysis compares two different objects and reports on the differences between them; for example, two text files can be compared to find all differences. Garfinkel has described a similar model specifically for DF.

Garfinkel has argued that (non-digital) differential analysis is already commonly practiced across several domains, such as reverse engineering and network engineering. He has pointed out that a general model consisting of three items constitutes a valid differential analysis: a *baseline image*, a *final image*, and a *differencing strategy*. He has taken these ideas and implemented them into a differential model specifically for DF.

Briefly, the model operates by first defining metadata for various features of a filetype and then determining the steps required to transform between two states containing those features. Features include a byte sequence, location, name, and timestamps. With these, the authors wrote tools ranging from a disk image difference finder to a pcap difference finder. In addition to simple tools, this type of analysis provides the basis for various high-level use cases: malware discovery, insider threat identification, and pattern-of-life definition.

This model provides a high-level description of what to do when conducting a differential analysis. The authors have demonstrated that their work could be useful as a basis for further development of differential tools, and they have included definitions for developing metadata tags to facilitate differential processing. However, this model falls short of providing the analyst with a *practical* means of specifying a forensic computation.

2.2.3 High Level Models

Researchers have also proposed high-level models. These consist of guidelines, best practices, and other models that cannot resolve to a specific computation.

Cognitive task model

As opposed to models that are too low-level, mathematical, or impractical, Roussev has proposed [77, Ch. 3] that adoption of a cognitive task model describing the *work* of the intelligence analysts [71] 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 usability), the resulting description is not formal and does not address the concerns of integrity and reproducibility.

Best Practices

At a somewhat lower level of abstraction, we find a bevy of procedural models that deal with specific scenarios, such as disk acquisition, Android forensics, and social media analysis. Those models are, in effect, efforts to establish best practice guides for practitioners. For example, the Scientific Working Group on Digital Evidence offers numerous best practice models for certain situations: examining magnetic card readers, cell site analysis, and image authentication, among others [83].

Looking specifically at image authentication best practices, the authors have described the methodology that should be employed when reviewing a digital photograph for evidence. These (paraphrased and non-inclusive) steps [1] include:

1. Preserving the original image,
2. Extraction of metadata (camera model, camera settings, GPS, etc),
3. Analysis of file packing method (hex-level headers, EXIF, etc),
4. Examination of noise within the image,
5. Examination the image contents (scaling of objects, evidence of staging, etc)

[1] The full set of suggested steps respective to Image Authentication methodology is included in Appendix A.3

However, best practices such as these fall short of being formal, generic, and reliably reproducible.

2.3 Digital Forensics Languages

The idea of creating a DSL for DF has been *tangentially* explored by prior researchers. Their research has addressed symptoms, but not the root cause, of problems across the field. The ultimate root problem is the lack of any practical and standardized description of the forensic computation. We review their work below but ultimately find they are unsuitable for general purpose usage or are not computationally specific.

2.3.1 *DERRIC* - A (narrow) data language for Digital Forensics

Perhaps the most explicit attempt at creating a DSL for DF is the *DERRIC* project at the *Netherlands Forensic Institute* [5].

DERRIC introduces a language to declaratively specify data structures, allowing for data processing to occur upon multiple variants of data types. One problem the researchers aim to solve is the inability for investigators to deal with multiple encodings of a single datatype. Furthermore, they aim to use *DERRIC* to handle reverse engineering of multiple valid file format configurations and encodings.

Specifically, the *DERRIC* language is (mostly) human-readable and uses keywords relevant to data description. For example, we have included a *DERRIC* file description for JPEG files in the appendix A.4.

With this type of general description, tools can be cognizant of *specific fields* within various formats (or versions) of file types (ex: version, endianess, etc). Tools and investigators can use these specific fields to describe, analyze, and act on *abstractions* of data, which in turn, decouples tools from specific formats.

While this is a limited domain-specific language (specifically, it is a data definition language), it does not address the concerns we have identified in chapter 1. It is written in a human-readable, but unwieldy, format. In short, *DERRIC* is too verbose, is not natural, and does not grant end-user analysts the power to specify the specific computation they wish to execute; rather, it allows for (expert) programmers to specify fields that can augment the processing of files by tools.

2.3.2 *DEX* - Digital evidence exchange

Levine et al. have introduced a library to logically describe different file formats [49]. Their main purpose is to introduce an open format for digital evidence provenance, both for description and comparison of prices of evidence. Furthermore, their approach allows for tool interoperability and result reproducibility. While the library falls short of a formal DSL, some goals are similar.

The library – *Digital evidence exchange (DEX)* – consists of an XML based file description for individual filetypes. Similar to *DERRIC* 2.3.1, this approach allows end-users to computationally describe byte-level file formats with the goal of allowing tools to operate on them (a data description language). However, *DEX* allows for an essential feature - *tool interoperability* - and allows for building toolchains. Furthermore, *DEX* allows for reproducibility and evidence provenance. On the surface, this approach is promising and indeed, meets several necessary demands. However, *DEX* falls short of providing a full solution.

One shortcoming is the difficulty of use – *DEX* file descriptions require a rather complicated XML structure (see Appendix A.2 for a sample code listing). Further, it is released as a Java library, and its integration with non-Java based tools is unclear. Finally, for tool interoperability to occur, individual tool maintainers would need to agree on and retrofit their tools towards a standard output format. Nevertheless, certain ideas within this research are useful to extract and utilize, such as enabling tool interoperability.

2.3.3 Formalization of Computer I/O - Hadley Model

In this work [36], Gerber et al. have described a model that encapsulates input and output operations between peripheral devices specifically for a forensic investigator. They have modeled their work after the seven-layer OSI model [66] and have proposed rules pertaining to data exchange between computing layers – where layers are components such as USB disks, HDD platters or heads, and system buses. They have proposed a functional language to describe translations between data formats at differing levels of abstraction. The ultimate goal of this model is to log verifiable transactions that occur at the *input/output* layer.

However, this approach is unrealistic – the authors have not explained how to handle the volume of inputs and outputs that would undoubtedly consume the system. For example, the act of saving a large file to disk could incur thousands of individual writes of sectors to a hard disk. Furthermore, this model does not address networking I/O operations. However, the research does touch on a point with which we agree – there is a need for a language to describe the forensic computation.

2.3.4 XIRAF - XML indexing and querying for DF

In this research, Alink et al. of the *Netherlands Forensic Institute* NFI, the authors have discussed an XML approach to managing and querying forensic data gleaned from digital evidence, called the XML information retrieval approach to digital forensics (XIRAF) [2]. This research involves three main elements: "a clean separation between feature extraction and analysis, a single,XML-based output format for forensic analysis tools, the user of XML database technology for studying and querying the XML output of analysis tools". We discuss each of these elements below.

Separation of *extraction* from *analysis* is a key idea. As the authors have pointed out, allowing extraction tools to exist interdependent of analysis tools largely enables the automation of feature extraction.

On the other hand, building an infrastructure revolving around an XML specification has some (surmountable) issues: First, XML is extremely verbose and unwieldy. The format is prone to hard-to-find typographical errors. Further, while XML *can* be as efficient as, for instance, JSON, less verbose format parsing will naturally reduce overhead.

Finally, querying an XML-based database requires knowledge of a difficult-to-use programming language - XQuery [2]. The authors have acknowledged its difficulty and have developed a separate GUI interface to hide the complication from the end user. We believe that the query language should be well within the grasp of a non-technical investigator.

[2] XQuery-https://www.w3.org/TR/xquery-31

Discussion of *XIRAF* would not be complete without discussing its scalability [101]. Not only is the dataset one that can be queried, but it was also designed to provide DF *"as a service."* Before replacement by *Hansken* (discussed in section 2.4.6), *XIRAF* was the NFI's primary repository for digital forensic investigations and provided a platform for petabytes worth of target data[102].

Despite shortcomings, such as closed source and opaque querying constructs, *XIRAF* is a promising start into the idea of a general data store and querying architecture for DF.

2.3.5 DFXML

Garfinkel has also researched and proposed an XML-based digital forensic file description language called *DFXML* [31].

As opposed to *XIRAF*, *DFXML* specifically aims to enable tool interoperability and not focus on result storage. Garfinkel argues that contemporary forensic tools are monolithic – designed to ingest a limited set of specific file formats and produce a limited set of output types (such as reports). The opposite paradigm (a Unix tool style) of small tools performing specific tasks that can be chained together to accomplish great tasks is what DF requires.

DFXML works by having experts review filetype formats and generate an XML-based description of the file. Features of this file vary between filetypes but commonly include byte runs of file data, hash information, and relevant timestamps. Moreover, the authors have provided an application programming interface (API) that ingests this "formatting" file as well as the target forensic data. This API is capable of being combined with further forensic tooling.

However, *DFXML* does not satisfy all of our concerns (nor does it intend to). Foremost, there is no easy or natural way to express the forensic computation. Second, while the research promises tool interoperability, it is not clear how non-Python tools would interact with the Python-based API. Finally, an XML-based file representation is not ideal.

In conclusion, *DFXML* is solely a data interchange format for a data forensics API. While iIt is not a full domain-specific language DSL, but it does encapsulate some research we can utilize. For example, interior file formatting regarding the best representation of time and GPS coordinates, among other things, has been researched. Furthermore, using small individual tools collectively is a key idea we keep in mind and consider during our research.

2.3.6 Digital Forensics Ontologies

A brief discussion regarding development of digital forensic ontologies is warranted. There are a few efforts currently underway to categorically define the vocabulary to be used in digital forensics (*DIALOG* [45], *CybOX* [12], and *UCO* [13]). In each instance, researchers define the relation between various forensic objects and offer a standard vocabulary for each.

These researchers have identified the same problem that we (and others) are looking to eliminate: there is currently no formal way in which to share forensic data. Current practices are informal and imprecise. However, unlike other researchers, we aim to address this with a DSL, which would necessarily incorporate common syntax and semantics.

2.3.7 SEPSES - logging with common vocabulary

Recent work has pointed out that the growing number of log sources, when combined with the volume of logs encountered in enterprise-sized environments, creates a complicated situation for security analysts [25]. The authors have argued that despite the advancement of some logging tools, the analyst is ultimately responsible for creating confusing regular expressions to pick out desired items of data from log streams (which are often verbose, redundant, or poorly structured). They have also provided a semantic model used within a deployable platform ("*SEPSES*") which relies on using their proposed vocabulary (an evolution of MITRE's *CEE*[60]) to query logged events from a central (or 'linked') datastore.

On the one hand, *SEPSES* addresses a specific subset of what other ontologies aim to do: it provides analysts with a common vocabulary to query *logs*. On the other hand, we point out that it does not promise consumption of data from *any* source - such as directly querying NTFS images, memory dumps, or network captures.

2.3.8 Roussev - a DSL for Digital Forensics

In 2015, Roussev first proposed the idea for a domain-specific language for digital forensics [76], citing several reasons for its necessity. First, from a legal standpoint, third-party verification of the investigation is necessary, and current methods are impractical, especially in the face of volume growth. Second, scientifically, the lack of a common language describing forensic processing hinders research efforts. Third, educationally, the lack of a language means that students are often trained on specific tools and products with GUIs – rather than learning the *process*. Finally, from a professional point of view, vendors often provide features that users do not ask for because they have no means to express their needs. In short, Roussev has demonstrated that a DSL built for the DF community has far-reaching benefits.

We build on the ideas initially sketched in this paper, and we further discuss their evolution in Chapter 3.

2.3.9 Summary of proposed models and languages

In summary, we chart related research in Figure 2.1. Clearly, *Nugget* is the only solution that provides both a *practical* and *computationally specific* foundation for the future of digital forensics.

2.4 Scaling Digital Forensics

As discussed in Chapter 1, the exponential growth in data volume in investigations is a serious problem. Indeed, it has been recognized as a future concern for at least a decade. However, as the volume of data is now overwhelming commodity hardware, it has risen to a pressing *present* concern. Several researchers have identified this issue, some of whom offer tentative solutions, and we summarize their work here.

2.4.1 Initial discussions

At least as early as 2004, Roussev et al. discussed the need for a *distributed* digital forensics solution in *Breaking the Performance Wall: The Case for Distributed Digital Forensics* [80]. In this research, Roussev et al. have pointed out at least three main reasons for building distributed forensics tools:

1. Growth of High-Capacity Storage Devices

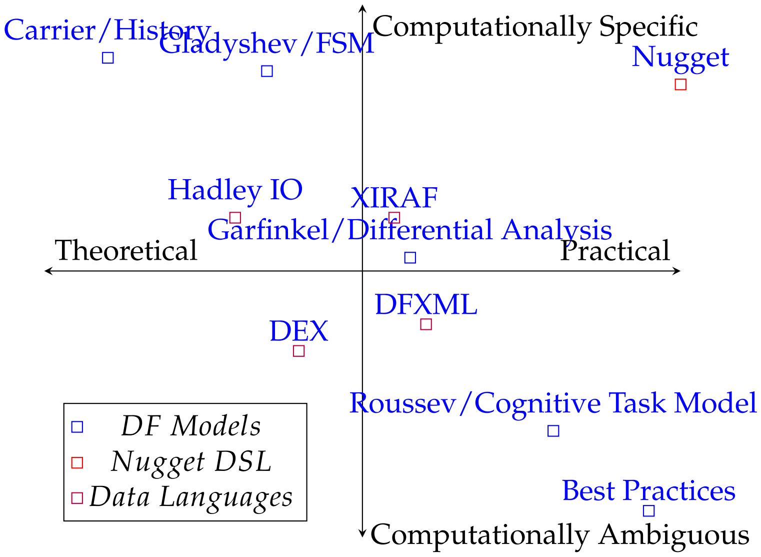


FIGURE 2.1: Comparison of conceptual models in digital forensics

2. Growth of I/O speed vs. capacity
3. Increased sophistication of digital forensics analysis

This research, however, has aged – after 15 years, no significant implementation has been realized. Fifteen years is 13 full cycles of *Kryder's Law* [3]; When the paper was originally written, a commodity laptop had a 40-GB hard drive [4]. Today's commodity laptop has a 1TB *solid state* drive [5].

Despite hardware advancements, widespread deployment of a distributed system has been hindered. Possible causes include deployment difficulty, (seemingly) high-performance workstations, and stagnant IO improvements. As storage capacities continue to grow, even high-performance workstations will become unsuitable for forensic investigations [79]. Further, IO improvements will remove a bottleneck, and force the realization that the current generation of non-distributed tasks are too slow to keep up with a significant volume of data. Finally, we discuss a solution in chapters 3 and 4 which reduces the difficulty of integrating scalability to standard digital forensic tools, paving the way for widespread adoption.

A growing need. To quantitatively illustrate the recognized need for scalable forensics, we turn to the U.S. Department of Justice and reports of outstanding service requests. In 2014, an audit report [99] indicated that across the FBI's 16 Regional Computer Forensic Laboratory (RCFL) units, there were 1,566 open requests as of August 2014. Of these, almost 60% were over 90 days outstanding: 381 (24.3%) were between 91 and 180 days old, 290 (18.5%) were between 6 and 12 months old, and 262 (16.7%) were over a year old. A more recent audit of

[3] "Inside of a decade and a half, hard disks had increased their capacity 1,000-fold, a rate that Intel founder Gordon Moore himself has called "flabbergasting." Kryder's Law" [48]

[4] data retrieved from https://support.apple.com/kb/SP81

[5] data retrieved from https://www.apple.com/macbook-pro/specs/

the New Jersey RCFL has suggested that in 2016, 194 service requests were not closed within 60 days, including 39 that were more than a year old.

Consequences of backlogs. Shaw et al. [85] have pointed out the possible consequences of delayed digital investigations: suspects can commit suicide, be denied access to family members (for example, after allegations of abuse), or face reduced sentencing because of the length of time spent waiting for the investigation to conclude. There are less serious considerations as well: during investigations where computers are seized, family members have no access to the data on the computer – such as family pictures and financial data. In cases where serious contraband is found, the computer may never be returned as remnant artifacts could be inadvertently returned to a family. While these concerns will always exist, investigators are facing *more* cases which each contain *more* data, exacerbating these issues.

2.4.2 Hadoop and Sleuthkit integration

One proposed solution centered upon utilization of *Hadoop*- a collection of open source utilities commonly used in 'big data' processing. Overall, *Hadoop* is widely adopted and highly regarded for its MapReduce capabilities. In 2012, Stewart attempted to fit digital forensics on top of a *Hadoop* model [91]. Unfortunately, he published no throughput numbers and, furthermore, the project appears abandoned. However, older research [81] indicates that the high latency introduced by disk access during the two-phase map-reduce processing would make a solution built on *Hadoop* unsuitable for forensics at scale.

2.4.3 Pig, Hadoop, and Sawzall

Apache Pig [96] is a dataflow language designed to describe the incremental steps in the processing of large data sets. Its primary users are data researchers and programmers, so it is designed to support interactive exploration of "big data." The actual computation is translated into (Java) *MapReduce* jobs that are execute on a *Hadoop* [95] cluster. The inspiration for *pig* comes from *Sawzall*–a similar language [70] developed at Google to serve as an abstraction layer over the company's *MapReduce* infrastructure. While these projects are not specific to DF, they are directly relatable as they are built specifically for large-scale data processing.

2.4.4 Distributed Environment for Large-scale Investigations *DELV*- Distributed Forensics

Richard and Roussev have proposed a distributed framework, Distributed Environment for Large-scale Investigations (DELV), built specifically to address concerns in digital forensics [74]. Specifically, they describe an experimental architecture consisting of a "coordinator" and "slaves"; the former distributes individual work items to the latter based on desired operations specified by an analyst. Results are then saved in a collective data store. This architecture emphasizes significant amounts of RAM, as analysis and reporting executes on cached data. Other features of this style architecture includes support for multi-user tools, support for an interactive user-interface during processing, and support for hardware expansion.

Conversely, Ayers [3] has described a system to minimize reliance on RAM. This directly competing evolution has many of the same earmarks – such as distributed workers, a coordinator, and cluster deployment – but argues that an investigation cannot assume large amounts of available memory (because of the size of the investigation) to store processing data; therefore, "the observed improvement of 18–89 times [of FTK in DELV] is unlikely to be realized in the general case where local or centralized disks would be required for evidence storage." However, the authors have overlooked the eventual freeing of resources and the ability to store evidence in SSDs or other fast media, which, even with redundancy (RAID), could keep up with total *DELV* throughput. In short, it is safe to use large amounts of memory for distributed forensics.

2.4.5 GRR - Distributed Incident Response

The Google Rapid Response Framework (*GRR*) was first introduced in 2011 by Cohen et al. [20] and has been studied in depth by others [14]. It is an open source, agent-based forensics tool aimed at providing live system data across multiple platforms. *GRR*'s architecture consists of a one-to-many relationship between a *server* and *client* computers running agents. When an investigator wishes to execute a forensic task, he initiates a *flow*, which can be distributed to many workers. This framework provides many valuable features but is not readily interoperable with other security tools. We discuss this problem, as well as provide an in-depth look at *GRR*, in Section 4.

2.4.6 *Hansken* project

We end our discussion on scalable DF with somewhat unique and relatively recent scaling research – the *Hansken* project *Netherlands Forensic Institute* [101] [102]. *Hansken*, an evolution of a prior distributed system, is advertised as "Digital Forensics as a Service". A large focus of its development centers on the investigative process. That is, how digital analysts interact with detectives. For example, the authors discuss how to process raw data to answer unique, novel, or ambiguous questions from case detectives, not digital analysts (ex: "search for the name Pete", "get all information related to drugs", etc).

By focusing on the investigative process rather than blindly seeking throughput of data through tools, *Hansken* can significantly speed up investigations. Investigators can, for example, use a web portal to view digital material instantaneously. Overall, the publication cites that its system facilitated hundreds of criminal investigations supporting over 1,000 detectives over the course of 3 years.

“Next-generation” digital investigation systems should, likewise, keep the entire investigative process in mind during research and development. This holistic approach requires a collective, *scientific* model.

2.5 Summary

We conducted an in-depth review of the field of DF methodologies and models, including the need for and implementation of scalable and custom language solutions. We can state with confidence that although tangential attempts have been made to cure symptoms with models and related implementations, no suitable DSL exists for DF. Furthermore, the benefits of this approach are far-reaching.

In Chapter 3, we thoroughly discuss the implications of such a language, as well as the potential impact of an integrated distributed architecture.

Chapter 3

Proposed Methodology

3.1 Introduction

As previously stated, the purpose of this research is to establish a *scientific* mechanism that addresses the current problems of reproducibility, verifiability, interoperability, and scalability of tools and processes across the field of DF. This mechanism should be a *layer* in the forensic architecture — it is an intermediary between the forensic *query* and the forensic *tool*. For further context, our research claims are listed again below:

1. Provide digital forensic investigators the means to specify a digital forensics computation that is both practical and formal;
2. Provide digital forensic investigators a means for reproducing digital forensic investigations;
3. Provide an abstract layer of communication suitable for use between forensic analysts, law enforcement, and tool developers;
4. Perform a mechanism by which digital forensic tools can be benchmarked;
5. Provide external entities (e.g., NIST) a means to validate digital forensics tools;
6. Provide educators a tool-agnostic medium to teach digital forensics processes;
7. Provide a framework enabling interoperability between digital forensic tools;
8. Create a flexible, scalable, and container-based runtime to demonstrate effective usage of *Nugget* queries.

In this chapter, we outline our research methodologies. Briefly, we propose creating a DSL for DF. We integrate this language with a distributed computing architecture, intending to nullify present and future issues related to data volume growth and data acquisition speed. Finally, we integrate our solution into several contemporary security tools and provide a mechanism for simplifying their integration.

We organize the remainder of this chapter as follows. First, we present the background of DSLs, including our requirements for a forensics-specific DSL. Then, we discuss the forensics-specific distributed computing architecture, whose ultimate goal is to be able to "throw more hardware" at the processing. Next, we discuss the metrics we must capture for solution validation, and we close the chapter with a conclusion and set the stage for 4, Results.

3.2 Develop a language

3.2.1 What is a DSL?

A Domain Specific Language (DSL) *is a computer programming language of limited expressiveness focused on a particular domain* [29]. The following are the critical components of this definition:

1. *Formality*. A DSL is a formal language, with established grammar and semantics, that translates into *executable* code.
2. *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.
3. *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*.
4. *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 summary, a DSL provides domain experts with the constructs necessary to describe a problem or solution succinctly and efficiently, and it 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 [37].

Think SQL Before the release of Structured Query Language (SQL), the access of information held within databases was irregular and unconventional. In 1970, Codd of IBM research defined the relational data model and went on to lay the foundation for the SQL with low level relational calculus constructs ([19], [18]). "SEQUEL" came about in 1974 [17] and it evolved into the pervasive data-access language used across the world today.

Analysis of SQL has modeled the design constraints necessary for a successful DSL. Chamberlin, SQL's original creator, states the following [16]:

> "The principal goals that influenced the design of SQL were as follows:
>
> 1. SQL is a high-level, non-procedural language intended for processing by an optimizing compiler. It is designed to be equivalent in expressive power to the relational query languages originally proposed by Codd.
> 2. SQL is intended to be accessible to users without formal training in mathematics or computer programming. It's design is to consume keyboard-based input. Therefore it is framed in familiar English keywords and avoids specialized mathematical concepts or symbols.
> 3. SQL attempts to unify data query and update with database administration tasks such as creating and modifying tables and views, controlling access to data, and defining constraints to protect database integrity. In pre-relational database systems, these tasks were usually performed by specialized database

administrators and required shutting down and re-configuring the database. By building administrative functions into the query language, SQL helps to eliminate the database administrator as a choke point in application development.

4. SQL is designed for use in both decision support and online transaction processing environments. The former environment requires the processing of complex queries, usually executed infrequently but accessing large amounts of data. The latter environment requires high-performance execution of parameterized transactions, repeated frequently but accessing (and often updating) small amounts of data. Both end-user interfaces and application programming interfaces are necessary to support this spectrum of usage."

Clearly, SQL exemplifies the key tenets of DSL design.

SQL is insufficient for DF. One ostensible solution is to bend the current technologies to the needs of the digital forensics community. Unfortunately, this will not work for the following reasons:

1. **Data Retrieval** Forensic investigations begin with the critical data retrieval and recovery phase. This includes processes such as parsing an NTFS disk image. There is no clear way to implement such a process within the design constraints of SQL.
2. **Standardized data model** To use SQL effectively as a forensics language, we would need to define a specific, *standard* data model — essentially, a set of tables — to represent all cases. Considering the significant effort and minimal success of less ambitious standardization efforts (and clear disincentives for vendors), we see no realistic way in which this would work. Furthermore, advanced SQL queries (involving multiple table joins) are advanced queries and thereby defeat the primary purpose of a forensic language.
3. **Graph Relationships** Querying relationships and dependencies between artifacts will be a primary use case of this language. This leads to non-intuitive, multi-table queries, and it results in poor performance at scale. "Big data" companies, such as *Google*, *Facebook*, and *Twitter*, maintain performance by matching collection types to different types of data stores. A similar approach for DF would require abstracting away internal representations of data — the exact opposite of what a SQL-centric solution would do.

Internal vs. external

Domain-specific languages fit within one of two broad categories: *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* [64], have advanced built-in support for developing internal DSL. The major disadvantages are that a) valid syntax for the DSL *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, on the other hand, 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 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. In contrast, external DSLs are needed for more general solutions, such as for host-language independence.

One illustration of the trade-off is the development of *Puppet*, which initially had a *Ruby DSL*, and was later abandoned in favor of an external solution [72].

For a DF DSL. For our purpose, an external DSL is preferable to an internal one. Our target user is someone who understands the *process*, but not necessarily all background technology (for example, law enforcement officers who have never been exposed to programming). Therefore we must make every effort to introduce the DSL as natural expressions, and for natural syntax and semantics, the language *must* be free of host language constraints.

3.2.2 Developing a DSL now

Viewed in a narrow light, the development of a DSL is costly. It requires significant knowledge of both programming techniques and the particular domain in question. Furthermore, when a DSL reaches a large user base, there should be an expectation for upkeep costs as developers fix bugs, training materials are produced, and new features are introduced [58]. Finally, performance *could* be a concern as it is another layer of computing. In this section, we briefly justify the need for developing a comprehensive DSL for DF.

Bridge the semantic gap. A well-designed and efficiently implemented DSL can dramatically expand the number of users who can autonomously solve problems within their domain of expertise, especially problems that previously presented significant technical hurdles. This major usability gain is possible because DSLs are concise [37], 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 DF as investigators work with increasingly complex targets and cannot be expected to understand *in depth* the technical implementation of the tools utilized. While they still need to understand the methods conceptually and be familiar with the reliability and error characteristics of the methodology, 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: first, (mostly proprietary) integrated forensic environments that provide a point-and-click interface and second, 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, thereby making it impractical to test and validate them; specialized tools provide better visibility but require custom integration, which is 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 to address the traditional tension between proprietary implementations and the needs both to test and to independently establish the validity of tools via third-party testing. In this scenario, a vendor only needs 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 the National Institute for Science and Technology (NIST), could perform an independent test, which would go a long way towards alleviating reliability concerns.

Utilize mature DSL development tools. The current time and conditions are ideal for developing a new DSL; after four decades of evolution, language development tools are mature and reliable, and modern integrated development environments (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.

3.2.3 Necessary attributes of a Forensic DSL

After consideration of the related work and literature review, we consider the following to be key and necessary criteria for an effective and adoptable DF DSL:

1. *Formal* - As with any DSL, statements must resolve into explicit computations.
2. *Natural* - Again, as with any DSL, statements must be natural to the subject matter expert who is expected to use the DSL.
3. *Declarative syntax* - Declarative languages, as opposed to procedural ones, allow the user to specify *what* needs to be done, as opposed to *how* to do it. This

allows the analyst to emphasize the forensic processes while abstracting away technical details of execution.

4. *Delayed execution* - Until results are required for additional computation or presentation, or until the analyst *specifies* immediate execution, the runtime should not execute computations. This allows analysts to build and fix complicated queries prior to (possibly) lengthy execution times.
5. *Tool agnostic* - Tools should not be baked into the DSL code; rather, they should be 'plug and play,' whereby tools of similar function can be substituted for one another. This allows for a clean division of code while simultaneously providing confidence in the results of competing tools if their results match.
6. *Extensible* - Given the nature of the DF community, wherein hundreds of individual tools exist, the language must be easily extensible. That is, the DF community must be able to integrate their favorite tools into the language. We envision a build script which seamlessly integrates new functionality into the language, in addition to a public-driven repository encapsulating additional functionality in the form of optional plugins.
7. *Support big data/AI* - We can expect a growing fraction of the evidence to be sourced from online services rather than physical devices [77, Ch. 6]. As most tools are tied to the filesystem API [78], a DSL can provide a seamless transition to the new network-based acquisition processes. Volume growth will also necessitate the utilization of machine learning or 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.
8. *Streamline education* - Just as SQL allows relational database-related education and training courses to provide meaningful skills without understanding the database engine implementation, we expect a language such as this one to provide the medium for training competent investigators without overwhelming them with 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 more advanced processing would likely require more specialized training.

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 understand readily, while also providing a formal specification of the computation that needs to be carried out. Structured Query Language allowed for numerous competing implementations to co-exist, thereby allowing for fast development, optimized execution, and autonomous GUI development.

3.2.4 Describing a language

The formal description of programming languages has been a focused topic of computer science since at least the late 1950s. In 1959, John Backus first proposed the syntax describing a programming language called IAL (now known as ALGOL58) [4]. The metalanguage [1] he used to describe IAL eventually became known as Bakus-Naur Form, or *BNF*. Over time, helpful operators were added, resulting in *Extended* BNF, or EBNF.

EBNF. EBNF is the standard means of describing the grammar of formal languages [103, 84]. The essential concepts are those of *terminal symbols*, such as the literal numerals “1”, “2”, and “3”, and *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.

EBNF example. As an example, consider Listing 3.2.4. This *EBNF* description of a language represents a parser for a simple calculator, supporting just addition and subtraction. The "root" rule is an Operation, which consists of a number, followed by at least one combination of Symbols and Numbers (as with regex, one or more is denoted by the ’+’). A Symbol is then defined as the addition or subtraction (denoted by the literal ’+’, an ’or’ ’|’, and a subtraction literal ’-’). A Number is defined as one or more numerical digits, optionally followed by a decimal point and another set of one or more numerical digits.

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

Thus, valid inputs would include: *3 + 2* and *8.5-12.99*, but the parser would fail on input such as *.4 + 3*, and *8 * 4*.

Clearly, *EBNF* is a powerful description language and can describe complex languages with a rather uncomplicated set of rules.

Regular expressions are insufficient. One ostensible solution for describing a language is to utilize regular expression matching. While standard programming techniques rely on regular expressions for a variety of tasks, they are particularly ill-suited for any robust application, such as parsing DSL input. The most obvious reason is the lack of recursion support — any recursive levels would need to be manually added to the full expression. More importantly, however, is that a regular-expression based solution is not scalable. Any grammar with more than a few keywords would quickly become too fragile and impossible to maintain.

[1] metalanguage — a specialized form of language or set of symbols used when discussing or describing the structure of a language

3.3 Develop a Distributed Architecture

As evident from the literature review, there is no shortage of attempts to create a distributed architecture specifically for DF. Solutions exist that attempt to bend standard big-data solutions (Hadoop) on top of DF [91]; however, as discussed, they are ineffective because of disk latency [81]. Additional solutions also exist that require difficult setup, such as the *Hansken Project* [101], and others lack interoperability with today's tools, are closed source, or are vendor products.

We envision an open-source system that utilizes lightweight *containers* to handle individual tooling and task coordination, as is becoming common with cloud-based architectures.

3.3.1 Containers

Containers are virtualized environments that encapsulate the necessary capability to execute defined groups of processes. Given their lightweight nature and ease of deployment, they are quickly becoming the preferred method for building large-scale data processing systems. Despite their recent gain in popularity, they are not a new concept.

Origins. The original idea of encapsulation of runtimes can be traced to the *chroot* command, introduced in 4.2BSD [46][2]. FreeBSD4.0 then developed it into the *jail* command, which provides more elaborate containment mechanisms. "Jails are typically set up using one of two philosophies: either to constrain a specific application (possibly running with privilege), or to create a virtual system image running a variety of daemons and services" [46], and they remain the two basic usage scenarios to this day.

Linux quickly followed suit. Menage [56] introduced the term "generic process containers", the full vision of which took several years to implement. The initial steps of the implementation became known as control groups (`cgroups`) as part of Linux 2.6.24 [22]. The last major components needed were *user namespaces*, which allow per-process namespaces; they provide basic means to limit the visibility (and access) to resources, such as mountpoints, PID numbers, and network stack state. User namespaces became part of Linux 3.8 [52] (2013) and, combined with userland tooling developed by the *LXC* [53] project, provided the first out-of-the-box container deployment and management facilities.

As a direct result of the kernel mechanisms implemented to support *LXC*, multiple userspace tools have been developed and are quickly becoming popular. Of these,

[2] As an aside, Clifford Stoll was using 4.2BSD when he performed what is widely considered one of the first digital forensics investigations [92]

Docker [3] [57] is the most popular; however, several other projects – Google's *Kubernetes* [4] [7], *rkt* [5] [75], and *LXD* [6] [55] – also have strong industry backing. This has lead to a quick maturation of the technology and an active effort, the *Open Container Initiative* [7], to standardize both the image format and the runtime interface.These standards guarantee interoperability and set up a best-of-breed competition among the tools.

In summary, the idea of encapsulated runtimes is not a new concept. However, recent maturation of the concept has caused a rapid rise in their adoption as *containers*. Container instances encapsulate units of schedulable work, complete with all necessary resources – code, data, and configuration – to accomplish a specified task. In short, they offer the ideal solution for deploying disjoint, disparate digital forensic tools.

The remaining task, then, is to coordinate the multitude of containers we plan to deploy.

Container orchestration Recall the intention to deploy these containers across several servers. In order to coordinate disparate tools (containers) on distributed hardware, we turn to container orchestration. Before exploring orchestration, however, discussion on *coordination* is warranted – the question of what needs to be coordinated.

Deploying a distributed architecture incurs several challenges concerning coordination between nodes. Here, we summarize the most relevant:

1. *Service discovery* - When functions are deployed to separate machines in a complicated distributed architecture, a new worker node has no natural way of finding the networked assets it needs.
2. *Load balancing* - In a distributed architecture, individual nodes can quickly (and unexpectedly) become overloaded.
3. *Secrets and configuration* - When new nodes have no *a priori* knowledge of the running environment, sharing (encryption) secrets or other sensitive configurations becomes challenging.
4. *Health checks and restarting* - In a distributed architecture, nodes are constantly failing/timing out; a plan is necessary to handle restarting (or diagnosing) troubled nodes.
5. *Worker updates* - With the large uptime expected in production-level architectures, taking nodes offline for updates or maintenance must be a last resort and, some mechanism should be in place to quickly and seamlessly roll changes into production.

[3] docker.com

[4] kubernetes.io

[5] coreos.com/rkt

[6] linuxcontainers.org/lxd

[7] opencontainers.org

Competing products, such as Docker and Kubernetes, handle these challenges in similar ways: deployment of host-level background services. The host-level service (the orchestration) is responsible for some initial networking and service discovery, worker configuration, and restarting containers that have died. Load balancing is a relatively easy challenge to handle in its most basic form – the host service utilizes a round-robin or least-recent lookup over DNS; however, more effective (and complicated) schemes exist. Overall, coordination problems are difficult but surmountable.

3.4 Extend the Language to Integrate Multiple Third-party Tools

The "core," or delivered, functionality of the solution should be capable of executing standard forensic tools, such as The Sleuth Kit or Volatility. However, as mentioned, a primary design consideration of the solution should be third-party tool interoperability. We thus integrate and perform forensic operations utilizing disjoint DF tools whose individual design parameters are ostensibly incompatible with one another.

3.4.1 Application Programming Interfaces (APIs) and Interoperability

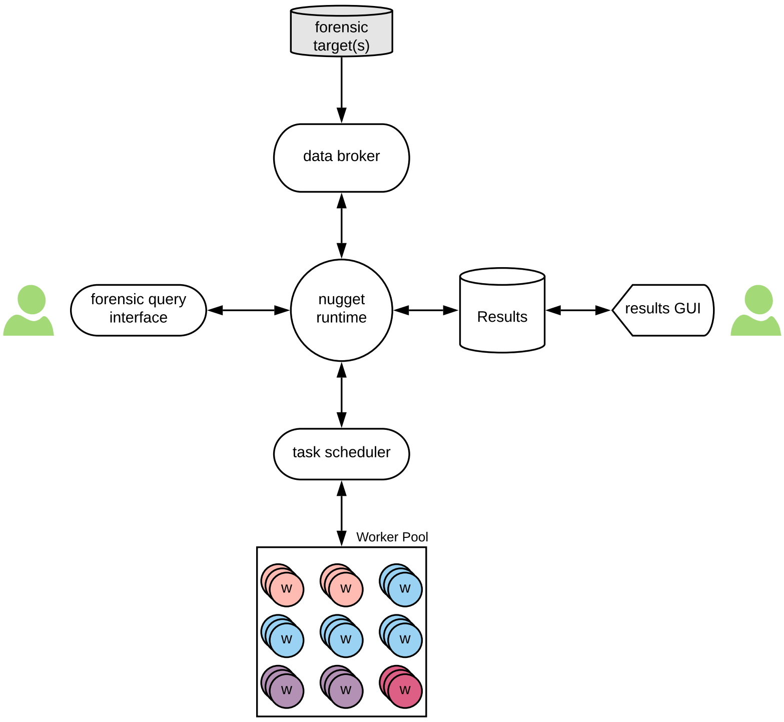
Most professional or production-level tools include an Application Program Interface (API), which allows external users (or programs) to request data or issue commands to the tool systematically. In general, APIs are prevalent across tools; however, they are all distinct.

The primary concern with distinct and uncoordinated APIs is their lack of compatibility. That is, just because two tools expose data query interfaces does not imply that the tools can be used together. In fact, nearly the only common factor between different tools' APIs is the use of a variation of an HTTP request mechanism. Indeed, the details of the HTTP implementation vary widely. As a result of the variations, multi-tool interoperability is impossible without establishing a translation mechanism.

As stated in Chapter 1, interoperability — the ability for tools to seamlessly work together — is a key feature that is currently missing in DF and indeed, across the greater security community. A common language would solve this issue by allowing users to define a common translation mechanism between tool APIs. The common translator, or "driver," could be published to the broader open-source DF community, allowing for an entire ecosystem to be built, which would ultimately result in a universal framework for security-related tools.

Proving it out. To prove the usefulness of tool interoperability, our solution should build API drivers for multiple tools. The solution should specifically demonstrate its capabilities in a scenario where the product of one tool is digested as data as consumption for another tool.

FIGURE 3.1: Integrated Architecture



3.5 Putting it all together

Now that we have an understanding of the various components of *Nugget* we can examine its integrated architecture, depicted in Figure 3.1.

This high level abstraction shows how the different parts of our research fit together. On the left side, *Nugget* queries are parsed. Their execution occurs in the *Nugget* runtime, which handles data ingestion as well as farming out tasks to an arbitrarily scalable number of workers, denoted by the worker pool. Results are fed back to the runtime, where it stores information into a scalable database.

A key feature of this architecture is its "plug and play" aspect. As long as the data interfaces conform to *Nugget* runtime's API, each component can be replaced with relative ease. It's also worth noting that thanks to the container-based design, additional types of DF tools can be easily added to the worker pool *without* the need for any substantial or architectural change.

More descriptive architectures for both *Nugget* and *SCARF* are included in Chapter 4.

3.6 Data Collection and Verification

All analyses shall be executed using the widely-adopted, standardized forensics corpora *M57* [32]. If necessary, follow-on analysis can be executed with other datasets, with a preference towards publicly accessible and popular forensic targets such as those given in [39].

About the forensic corpora. A brief explanation of the target datasets is warranted. The standard datasets – *corpora* – were developed in 2009 and, based on the high number of citations, have been heavily utilized for forensic research. Although they are somewhat older, they provide a well-established *standard* for new research to address their tools to. The corpora consist of several types of data, including network captures, memory (RAM) dumps, and HDD images. The data is organized into several different scenarios, such as the **M57-Patents** scenario, which simulates a law enforcement capture of various computing devices belonging to a small business suspected of illicit activities. In short, the corpora provide a peer-reviewed and highly sited dataset on which to test our research.

3.6.1 Use Cases

To demonstrate the capabilities of this system, we developed several use cases to offer a realistic context and provide a backdrop to support quantitative analysis. These use cases represent common evaluations of targeted forensic data.

Use case 1 - HDD evaluation for contraband hashes Arguably the most common forensic procedure involves extracting files from a volume and comparing their hashes to a list of known "contraband" hashes. For example, a suspected child abuser's computer will have its image files extracted, hashed, and compared to a list of hashes which are known to represent images depicting child pornography.

For our purposes, we will utilize a standard forensics scenario to simulate this case – the *M57-Patents* scenario[32]. This scenario represents law enforcement's investigation into a fictional small business after allegations its employees are storing illegal images on their work computers [8].

For this use case, the developed research will action the following:

1. Load the dataset into a standardized form,
2. Query this dataset,
3. Filter the results for image files,
4. Hash the image files,
5. Load a list of given hashes for 'illicit' images,
6. Compare and report the list of hashes from the dataset to the given ones

Use Case 2 - memory forensics for suspicious processes. To demonstrate the capabilities of incident response, the research must support memory forensics. Again,

[8] For the exercise, authors of this scenario have used pictures of cats to represent illicit imagery.

the **M57-Patents** scenario is ideal to demonstrate this capability. The goal of the investigator is to find evidence of an active keylogger on a target image.

To this end, the developed research will carry out the following:

1. Load the RAM into a form that can be queried;
2. Query this dataset for running processes;
3. Provide a filter mechanism for the analyst;
4. Report on the filtered processes to the analyst.

Use case 3 - network forensics for suspicious traffic Another major aspect of the investigation is the analysis of network traffic. The developed system will be able to integrate network analysis tools. To test this, the M57-Patents scenario again provides a suitable target. In this case, the investigator must determine whether any users of the network were attempting to exfiltrate proprietary company information to outside entities.

To demonstrate the capability to handle network analysis, the developed tool should perform the following

1. Load the network data into an internal format,
2. Provide a *runtime* filter mechanism for the analyst (dynamic, not static filters),
3. Report on the filtered network traffic to the analyst

3.7 Conclusion

In summary, we discussed potential methods to address the outlined problems presented in our introductory chapter and verified in our literature review in Chapter 2.

We theorized on design solutions for a scalable and tool-independent distributed computing solution to handle arbitrarily large forensic datasets. This includes the use of *containers* to easily integrate both existing and future toolsets.

We outlined high-level design requirements for a DF language. This language should be *clear and natural* to the forensic analyst, and individual statements should resolve to a specific computation. Furthermore, the runtime must be able to support compound statements that combine the results from one computation and feed them into another. An additional key feature should be background tasking and delayed execution – the analyst should not have to sit idle while computations are executing.

Finally, we discussed verification methods for *large* datasets and emphasize publishing results for executing third-party tools against them using a container-based solution.

In the next chapter, we discuss the development of these solutions.

Chapter 4

Results

In this chapter, we discuss the design and implementation of two systems built to address the concerns raised in Chapters 1 and 2, and whose requirements we outlined in Chapter 3. The goal is to build a human-readable language that is simultaneously capable of describing specific forensic computations. This requires a scalable, container-based distributed forensics framework as its runtime, which we detail after describing our language, named *Nugget*.

4.1 Nugget - A Digital Forensics Language

In Chapter 2, we presented a number of attempted solutions concerning verifiable or repeatable forensic tasks. Our language — named *Nugget* — aims to be the midpoint between best practices and purely mathematical models. Next, we define a formal model that works at the same level of abstraction as the analyst (similar to best practices), but that leads to an *unambiguous* computational description. A layout of competing conceptual models is illustrated in Figure 4.1.

More specifically, our proposed features demand the following:

1. *They must contain formality;*
2. *They must utilize declarative syntax;*
3. *They must be natural to the user;*
4. *They must have delayed execution;*
5. *They must be tool agnostic;*
6. *They must be extensible;*
7. *They must support big data/AI;*
8. *They must support tool validation;*
9. *They must streamline DF education.*

In this section, we discuss the implementation of these features and the reasons for certain design choices, and we thoroughly explain the user experience. However, first consider the following set of Nugget queries, which provide context for the upcoming discussions (Listing 4.1):

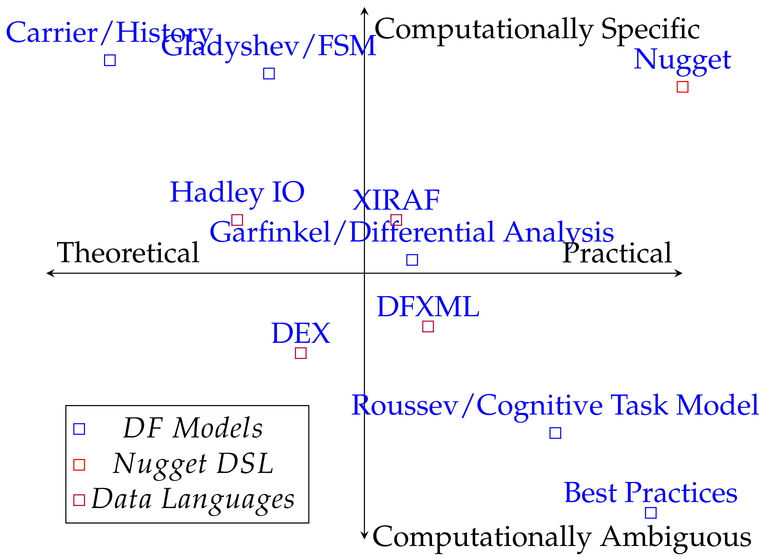


FIGURE 4.1: Comparison of conceptual models in digital forensics

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

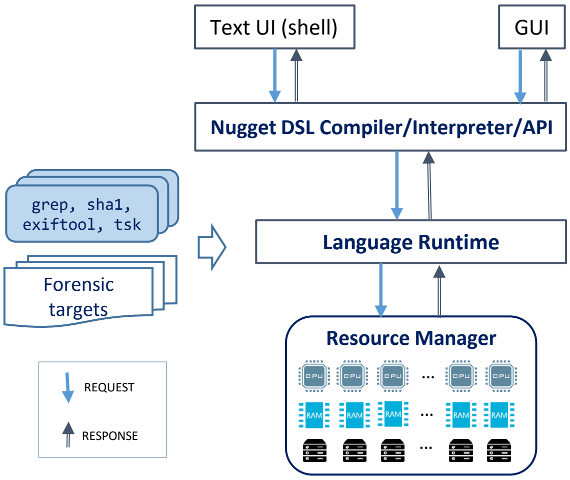
LISTING 4.1: NTFS file extraction, filtering (by size), and hashing

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 key points here are as follows:

1. the domain expert did not need to learn a general-purpose programming language in order to understand the intent of the query (and he or she could quickly learn to write similar queries);
2. this is a formal specification that can readily be translated into executable code;
3. the query only specifies what needs to be done and not how it should be performed; numerous possible implementations exist, including ones that employ the resource of a compute cluster or a (private) cloud service;
4. 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 understand readily, while also providing a formal

FIGURE 4.2: Layered forensic runtime architecture



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.

Next, we turn to an in-depth description of *Nugget*'s implementation.

4.1.1 Architecture - where *Nugget* fits

Nugget sits as an abstraction layer between the user (which could include programmatic users such as other programs and scripts) and the tool implementation. To handle tools, we need a resource manager such as *SCARF*. 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. See Figure 4.2

The language runtime maps the abstract representations of an operation, such as the hashing of a file, to an actual command 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 offers 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 DF is the lack of a means of clarity for users (forensic analysts and lab managers) to communicate functional and performance requirements to vendors. The main objectives of *Nugget* are a) to solve this problem by allowing analysts to specify queries they can reason about directly and b) to demand responsive solutions; it allows users to compare alternatives directly, and it creates best-of-breed competition among vendors. Conversely, a formal interface allows developers to have specific targets and to understand the needs of their customers better.

4.1.2 Implementation Details

In this section, we provide the reader with technical details of the implementation of *Nugget*. We include several relevant code snippets and examples; however, refer to the appendix for longer code snippets. The entire codebase is hosted on the author's public code repository [1]. Finally, we utilize our section on use cases (4.1.5) as an opportunity to explain further technical details.

Implementation language choice

Our implementation of *Nugget* is written in *Go* version 1.9 [97][2] – Google's static, strongly typed, structural programming language. Other implementation languages were considered, such as Java and Scala; however, Go was selected for three main reasons. First, *Go* is beginner-friendly and easy to use. While not a primary concern, easy extensibility is a desired trait of *Nugget*, and a popular, forgiving, and well-documented language lowers the learning curve. Second, *Go* is currently one of the most popular programming languages (with rapidly gaining popularity)[98], and as a direct corollary has gained widespread support by third-party tools – such as ANother Tool for Language Recognition (ANTLR) (discussed in more detail below). Finally, *Go*'s ability to generate binaries for numerous operating systems and architectures (e.g., Windows, Unix, ARM, and i386) is a desirable quality, especially when considering the future support for digital forensic triage on live, running systems.

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-, and 64-bit integers, strings, and dates, as well as several specialized data types, such as binary, hexadecimal, or base64 strings and standard kilo, mega, giga... notation for data units.

[1] https://github.com/cdstelly/nugget
[2] https://golang.org

Except for 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 connect the multiple operations in a single flow statement concisely.

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 4.1 to concisely explain their intended use.

Extractors. The main function of *extractors* is to shield the rest of the system from the particulars of the data format and method of ingest of the source. Extractors are operations that take a data source, such as a disk or RAM image, as input and produce collections of data items, such as files, processes, and packets, as output. 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.

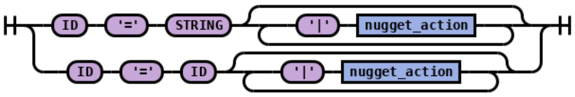
In our example, we used an NTFS extractor (from *The Sleuth Kit*), which parses a raw disk partition and extracts the file system metadata. In the specific case, we supplied two additional parameters, 63 and 512, which provide the starting block and block size, respectively. Each object is created with a set of known attributes, such as *name*, *size*, and creation time (*ctime*). 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. *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 2 of the example query filters out all files 1MB 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. *Transformers* are functions that produce output, such as a hash value, for each object in the input collection. On line 3, two values (a tuple) is produced based on the content of each file in the input set.

Serializers. *Serializers* are functions that produce an external representation for a collection; for example, *print* yields a textual representation suitable for shell environments. Furthermore, different versions of save can produce *json* or *xml* output suitable for storage. Subsequent work will integrate specialized evidence containers such as Advanced Forensics File framework (*AFF*) [33, 21].

FIGURE 4.3: A railroad diagram of assignment clauses



Optimization.

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, similar to those supported by SQL engines.

Grammar.

The standard means of describing the grammar of formal languages is the *Extended Backus-Naur Form* (EBNF) [103, 84], as detailed in Chapter 3.

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

```
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 various actions (where an action can be a transformer, a filter, or an extractor). Together, these quickly allow for complex queries to be described. Figure 4.3 provides a schematic of the above rules.

It is impractical to attempt to explain every *Nugget* clause exhaustively; however, it is essential 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, then a syntax error will occur at the *parser level*. This is a stricter scheme that allows for 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 a core design goal of *Nugget*. However, the benefit of this approach – error handling at the parsing level, and syntax checking *prior* to execution time – are more valuable from a usability perspective. Furthermore, 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) [67, 68] is an open-source parser generator. It takes an input grammar and, during a build step, produces the appropriate lexical and parser functions necessary to consume legal inputs. It also provides feedback to the user when erroneous input is encountered. This tool is capable of generating these functions in the following output languages: C++, C#, Go, Java, Python, JavaScript, and Swift.

The AST. As with other programming models, the result of the lexing operation is an AST, which is a useful structure that allows 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, a partial AST representing our sample Nugget code in Listing 4.2 is shown in Figure 4.4.

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

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

LISTING 4.2: Example *Nugget* query

Nugget grammar

There are at least two aspects to *Nugget*'s grammar that should be noted. First, that execution statements, such as extractions, transformers, or serializers are parsed as

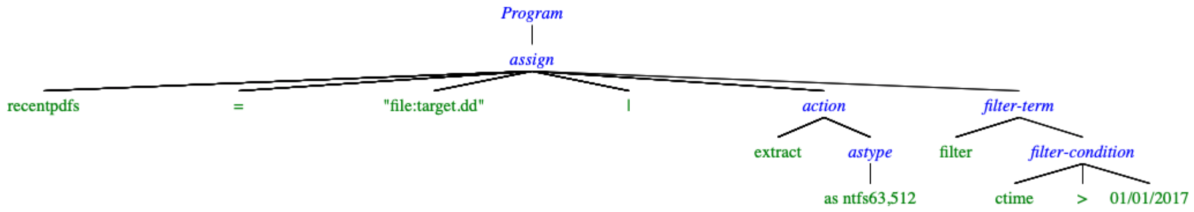


FIGURE 4.4: Sample AST generated by *Nugget* (partial)

a root group called an "action". While this leads to rather long groupings of terminals, it simplifies the execution code on the implementation side of the parser. This is partially how the AST is walked in ANTLR generated constructs, and partially due to Golang's lack of generics.

Second, we have included stand-in wildcards denoted by three percent signs (e.g., "%%%"). This gives our grammar building helpers a location to insert grammatical constructs when end-users are extending *Nugget*.

For *Nugget*'s full grammar and syntax, refer to its EBNF grammar listed in Appendix A.1.

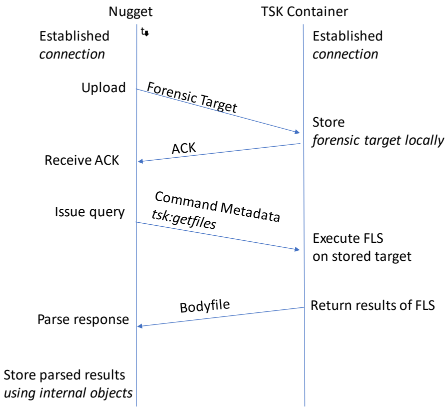
4.1.3 Runtime

As stated earlier, 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 *SCARF*, which employs a combination of RPCs and Linux containers via *Docker*.

Containers and *Docker*. Containers provide encapsulation of a process's runtime by granting access to the set of resources – CPU cores, RAM allocation, file systems, and networking – needed to perform a computational task. All containers share a standard OS kernel; however, by default, they are isolated from one another; it is also possible to set up sharing of resources where needed (e.g., software installations). Containers generally have a much smaller resource footprint than full-stack VMs, and the overhead to start up or shut down 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 straightforward to both containerize existing tools and to integrate them into *Nugget*. The specific containers we used here can thus be replaced by similar tools; alternatively, multiple versions of the computation could be run in parallel (e.g., a *Volatility* [51] [50] container and a *Rekall* [94] container) to increase confidence in the results.

FIGURE 4.5: Example RPC Protocol



Utilizing containerized tools with RPC. *Nugget* interacts with these service containers via RPCs. 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 RPCs issued by *Nugget* operate on the cache.

A sample protocol diagram is portrayed in Figure 4.5, 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 the section detailing *SCARF*, networked containers can be configured to distribute expensive forensic tasks, yielding a near-linear increase in throughput-per-container for many typical forensic tasks.

4.1.4 Extending Nugget

Nugget provides a mechanism that allows for it to be extended itself, thereby 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 as follows: 1) identify the type(s) of data that 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." This *build-nugget* tool generates and inserts into the defined *Nugget* grammar appropriate terminal nodes corresponding to the intended functionality. Furthermore, it generates template code for accessing the *Docker* container via an 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 provided in Listing 4.3, which illustrates how *sha1* is added to the grammar.

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

LISTING 4.3: Extending *Nugget*- sha1.json

4.1.5 Results

As discussed in our Methods chapter, we executed several test use cases to ascertain the viability of *Nugget*. We discuss the results below.

Use case 1 - HDD evaluation for contraband hashes

Analysis of hard drive partitions is performed via integration with *TSK* [8]. To understand this process, consider the sample *Nugget* code in Listing 4.4. 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, namely, files, which references a local file named *jo-1124.raw.* We must then instruct *Nugget* on 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 are the metadata for all files within the partition, and they are 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 explained, this will occur via RPC to a TSK container (Figure 4.5). 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 instead establishes a local set of filters that will be applied to any previous 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 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 how to run the forensic tool on the data provided via RPC methods.

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 4.5. It relies upon the results of *matched*, which in turn relies upon the results of *jpghashes*, etc., causing all dependencies to resolve.

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

LISTING 4.4: Join operation to find known files

LISTING 4.5: 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] ...
```

Use case 2 - memory forensics for suspicious processes

In our implementation, memory analysis is performed by *Volatility* [51]. To examine this more closely, we obtain the list of running processes by issuing a *pslist* command - a common initial step when inspecting memory.

In the first line of listing 4.6, 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*.

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

LISTING 4.6: Nugget Memory Analysis

LISTING 4.7: 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] ...
```

Use case 3 - network forensics for suspicious traffic

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 4.8.

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 with 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.

```
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 4.8: Nugget and HTTP

LISTING 4.9: GET Requests

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

4.2 SCAlabale Realtime Forensics - SCARF

We begin our results chapter with a review of our distributed computing solution for DF – named SCAlable Realtime Forensics (SCARF). After discussing architecture decisions, we explore backend software used to support the architecture. We present performance metrics of the system, including against forensic targets of various sizes and utilizing separate tools. We conclude our SCARF discussion with an explanation of the (lack of) a user experience.

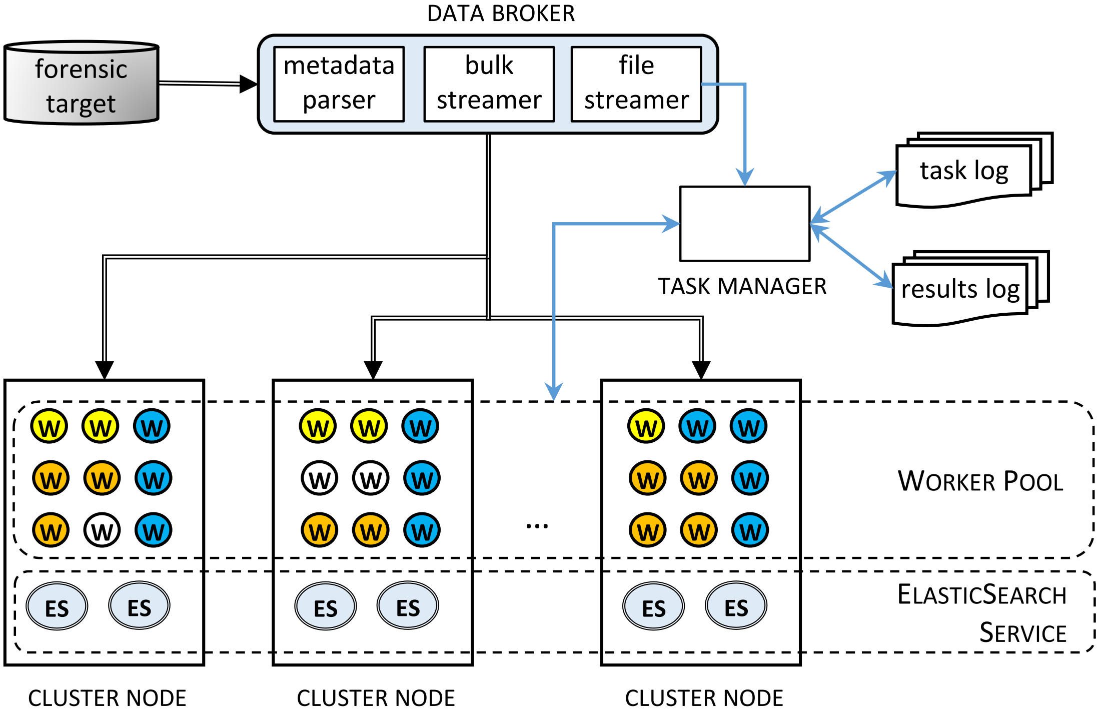


FIGURE 4.6: Architectural sketch of SCARF (simplified).

4.2.1 Architecture

Figure 4.6 provides an overview of the functional components and main data flows of SCARF. The *data broker* extracts the raw data from the forensic target and prepares it for streaming to the cluster nodes. The broker serves as an abstraction layer that decouples the processing of the data from its source format, such as a filesystem, RAM snapshot, or network capture. At present, we support two types of data access: *bulk streamer* and *file streamer*.

The *bulk streamer* provides sequential block-level access to an entire volume without attempting logical artifact reconstruction. It is suitable for tools such as *bulk_extractor* that function at the same level of abstraction and look for relatively small pieces of data.

The file streamer reconstructs files from block storage and transmits them as units of input data. It works by reading the file system data structures during initialization and reconstituting the files on the fly. We utilize a version of the *LOTA* approach described in [79] to optimize access times.

Task manager

The next major component is the *task manager*, which keeps persistent logs of task definitions and task completions. The definitions are generated by the data broker and depend on the set of available functions and the stream of data extracted from the target. As the simplest example, for every file identified on the target (by parsing the filesystem metadata), the broker will generate a *SHA1* task, which is added to the task definition log; we refer to it as the *task queue*.

The task manager uses *Apache Kafka* [3] to maintain its persistent logs and notify registered *data client* nodes of available tasks. Conversely, completed tasks are committed to the completed log. Although not a central point of this paper, we should emphasize that maintaining reliable logs of completed processing is *critical* to ensuring the integrity of the computation. At scale, errors in complex distributed systems occur with some regularity; experience demonstrates that it is infeasible to eliminate all possible source of failures. For example, in our case, out of the millions of spawned container instances, some will fail to execute.

The practical way in which to handle sporadic failures is to restart the computation; if the failure continues to occur, then it is systematic and needs to be debugged. We should note that failure can take the form of a task taking too long to execute, in which case it is better to terminate and restart it. Google's experience with map-reduce processing [24] suggests that a small fraction of the tasks tends to delay the overall completion of the processing. The solution to this is to run multiple instances of the same task and only take the result from the first one to finish.

The main point is that keeping reliable and detailed logs of the processing is necessary for both integrity and performance considerations.

Data clients

A *data client* is a container instance that receives a chunk of the forensic target and organizes the execution of *tasks* on the given portion. In order to optimize the workload, the chunks are distributed across all available data clients.

Upon creation, data clients generate a unique identifier and register to both the data broker and the task manager. As the data broker streams data from a forensic target to data clients, it simultaneously polls the task manager for outstanding tasks.

Tasks are added to the task manager with the unique identifier of the data client, and they can optionally be accompanied by file identifiers (in the case of a file streamer). Upon receipt, the data client becomes responsible for the execution of a given task. Furthermore, a *future* option can be added so that subsequently received files will also have that task executed on it. This feature allows tasks to *return results even as data is streamed from the original source*.

Task execution is *not* handled by the data client; it is farmed out to a dynamic pool of specialized containers, called *workers*. The rationale here is that storing data in memory and executing forensic tasks on the data are fundamentally different tasks. Moreover, separating worker containers from data clients allows for easy development and deployment of new types of workers.

[3] kafka.apache.org

Workers

A *worker* is a container instance that performs a specific task on the given piece of input data. Importantly, workers have no concept of files – they provide a remote procedure call (RPC) interface, through which they consume incoming data and produce a result in the form of a JSON string. Although the interface would benefit from the imposition of some additional structure on the I/O stream formats, this generic approach is quite flexible and allows easy creation of new types of workers.

The containerization of workers offers the ability to scale any task quickly. Containerization can be extended to easily *prioritize* tasks. That is, we can develop methods to scale containers based on server availability or task importance automatically. Within Docker, prioritization is easily implemented with the *docker service scale* command. (We would expect more complex algorithms to coordinate scaling as part of the underlying orchestration service eventually.)

As an illustration, the following will increase the number of containers running an *ExifTool* worker:

```
docker service scale exiftools=48
```

By consistently applying the above RPC approach, it is possible to automatically scale out any container, regardless of the specific computation performed.

Data clients are not aware of the number of, or network paths to, workers. Instead, network resolution (and rudimentary load balancing) is mediated by an internal DNS service. A lookup for an "exiftools" container will thus return a virtual IP to the least recently used container.

For our initial design, we have implemented a variety of workers representing common forensic tasks, each of which has different computational demands:

- *SHA1*: performing crypto hashing of data;
- *grep*: live regular expression search;
- *Tika*: text extraction with *Apache Tika*;
- *open_nsfw*: image classification using a trained *Caffe* deep neural network provided by Yahoo![4];
- *bulk_extractor*: feature extraction using regular expressions and verification; [35]
- *ExifTool*: metadata extraction from a files.

Results repository: ElasticSearch (ES)

As tasks complete, results are returned to the data client. After a batch of results is returned, or a predefined threshold is reached, the results are bundled and sent to a cluster of *ElasticSearch* (ES) nodes.

[4] https://github.com/yahoo/open_nsfw

The ES cluster nodes are deployed on the same hardware as the workers and provide a searchable database of the results from the tasks. It is integrated via its RESTful interface for storage, querying, and retrieval of data, and can dynamically scale to meet the incoming stream of results.

The ES modules are split between gateway, data, and master nodes. *Apache Tika* is also deployed within ES to enable the indexing of non-plaintext MIME types, such as PDF documents.

4.2.2 Architecture supporting software

In this section, we briefly discuss why we chose specific software over others. Some choices were made ahead of implementation time; however, other "best-fit" suites were discovered only after trial and error.

Docker vs. others. As mentioned in Chapter 3, Docker has many competitors. Desirable traits include ease of installation on a cluster, ease of use, networking support, and orchestration support; we also desired a widely adopted technology that could make other researchers comfortable extending this solution. With these in mind, we initially selected Google's Kubernetes *Kubernetes*[5] [7] container solution; however, we eventually discovered serious setbacks, which forced us to move to another solution.

The most severe issue we encountered was the installation of Kubernetes on our cluster at the Greater New Orleans Center for Information Assurance (GNOCIA). The *yum*-based package manager on the cluster is handled by a software suite, namely, *Bright Computing*'s *Cluster Manager* (CM)[6]. However, the extra layer of going through CM induced numerous compile, build, and runtime errors. Despite eventual installation across our cluster's nodes (via bypassing CM), the result was ultimately unstable.

Given the instability of our *Kubernetes* installation, we moved to a Docker-based solution. Nevertheless, *Kubernetes* is certainly worth future investigation as a long-term solution as it meets and exceeds our other requirements (indeed, it seems to be preferred over *Docker* in professional production deployments).

ElasticSearch vs. others. ElasticSearch[7] is a particularly dominant datastore. Its main competitor is Apache Solr[8]. Both solutions utilize Apache Lucene on the back end, but optimizations and indexing or querying are different. First, Solr primarily concerns itself with text-oriented query optimization, while ES provides better optimization of analytical queries. Second, both are technically open-source; however, a company behind ES ultimately has the ability to deny community contributions. Third, ElasticSearch configuration and querying are entirely centered on a JSON

[5] kubernetes.io

[6] http://www.brightcomputing.com/

[7] https://www.elastic.co/

[8] https://lucene.apache.org/solr/

REST API, which is more intuitive than Solr (although it is worth noting that Solr offers a less comprehensive REST API). Finally, both systems distribute easily, although ES is deployable on multiple cloud environments, while Solr is oriented towards SolrCloud.

In conclusion, while Solr and ES are similar, the latter is ultimately the more appropriate solution for ease of use.

Kafka vs. others Kafka was a particularly comfortable choice for our messaging queue system. Its primary features are performance, reliability, open source, and scalability. One (potential) downside is its reliance on yet another software suite – ZooKeeper – which handles cluster health. RabbitMQ was another alternative that we explored; however, the demonstrated throughput was insufficient for our needs.

4.2.3 Extending SCARF

Recall the design requirement for easily extending SCARF; indeed, that is the motivation behind the utilization of containers.

Containers, here treated as individual and distinct computational units, provide an ideal platform for extensibility – they can be scaled (launched and retired) according to available resources or priority. Most importantly, no modification is required for an existing tool in order to accomplish scalability. Furthermore, any forensic tool can be deployed within a container and thus to SCARF – as discussed previously, these workers simply take in data, perform an operation, and return a JSON string.

For most single-purpose tools, the process of incorporation into SCARF's processing fabric consists of three basic steps: 1) add a network layer for communication; 2) containerize the tool for scaling, and 3) invoking the tool. These steps are explained below.

[1] Build an RPC wrapper

First, we applied a small amount of wrapper code around the tool, with the goals of exposing a network path for the acquisition of forensic target data and providing a response mechanism. The wrapper provides the means to execute the existing tool on the provided input data and to return the results to SCARF. This can be accomplished in a variety of ways; we chose to employ Golang's built-in RPC library. Using an RPC allows for the results of a forensic operation to be returned while simultaneously providing a mechanism to acquire the forensic target.

Once an RPC function is invoked from SCARF, the given input is translated into a form that the tool can operate on, and the necessary command-line parameters are generated to launch the tool. In the developed examples, this involved changing two lines of code as *all* RPC, logging, and error checking code can be templated and abstracted away.

The overwhelming majority of the code in Listing 4.11 is quite generic and is useful as a template for the wrapping of similar tools. The two key variables are *toolPath* and *opts*, both of which will be dependent on the forensic tool in question.

[2] Containerize the tool

Second, the tool and wrapper must be implemented into a container image, a process we refer to as *containerization*. This requires the development of a container description file; in the case of Docker, it is called *Dockerfile* and provides a simple script for building the image. The starting point is a known system image, such as a clean operating system installation. The series of steps includes operations such as the installation of prerequisite software and setting the environment variables. The wrapper binary is also installed and set to auto-execute upon container creation. Listing 4.10 (wrapped to fit the column) illustrates a *Dockerfile* to build a *bulk_extractor* image.

The majority of commands to build the tool should be familiar to open source developers. The installation of pre-compiled binaries would be easier; however, in cases where building from source is desired, pre-existing build sequences can be added to a Dockerfile with minimal effort.

```
FROM ubuntu:trusty
MAINTAINER joe <joe@example.com>

RUN apt-get update
RUN apt-get install -y curl make g++ gcc  \
          netcat dnsutils vim flex   \
          libewf-dev libssl-dev wget

RUN wget http://digitalcorpora.org/downloads/bulk_ex
   tractor/bulk_extractor-1.5.5.tar.gz
RUN tar xvzf bulk_extractor-1.5.5.tar.gz
RUN cd bulk_extractor-1.5.5/ &&
    ./configure && make &&
    sudo make install

ADD bin/rpcserver /
ADD banner.txt    /

RUN mkdir -p /tmp/bulk_in/
RUN mkdir -p /tmp/bulk_out/

CMD [“/rpcserver”]
```

LISTING 4.10: Dockerfile for *bulk_extractor*

[3] Invoke the tool

Once the tool is containerized and exported as a network service, it is ready to be incorporated into the processing fabric. This is accomplished by adding a few lines of code within the *consumer*. The code to invoke *ExifTool* is shown on Listing 4.11:

```
func (t *RPC) Execute(args *Args, reply *string) error {
    toolPath := "/usr/bin/exiftool"

    // Setup the shell command to launch ExifTool
    // − indicates data will be read from STDIN
    opts := []string{"−’’}

    cmd := exec.Command(toolPath, opts)
    cmd.Stdin = bytes.NewReader(args.Data)
    var out bytes.Buffer
    cmd.Stdout = &out

    err := cmd.Run()
    fmt.Println(out.String())
    *reply = out.String()
    return err
}
```

LISTING 4.11: ExifTool RPC wrapper code

4.2.4 Performance and metrics

The effectiveness of the proposed framework relies on the ability to scale operations. To demonstrate scalability, we selected several tools to benchmark everyday forensic operations under a variety of conditions. Although a side effect of these examples is a processing rate that can (in some cases) keep up with SATA speeds, the critical factor is the relationship between throughput and the number of deployed containers. We demonstrate that an increased number of containers increases throughput. Therefore, additional hardware could be deployed, resulting in more containers, which leads to higher throughput.

4.2.5 Processing rates

The base configuration of our evaluation setup consisted of a cluster of four rack-mounted server machines connected to a commodity 10-GbE switch. Each box had 256 GB RAM and 24 2.6 GHz dual-threaded cores for a total of 96 physical cores and 192 logical ones. All nodes had a SATA-attached 1-TB SSD (Samsung 850 Pro), although this is largely irrelevant as all data for processing was handled in RAM. The observed throughput for bulk transfer over TCP connection was approximately 1

TABLE 4.1: SHA1 file hashing throughput (MB/s) vs. number of containers

Containers	4	12	24	48	96	192
MB/s	345	857	985	985	948	992

GB/s. At 5 years, the CPUs are three generations old and near the end of their lifecycle. The upside is that the results can be considered more representative as the hardware would be easily affordable for any lab.

The reference test data is a full 200GB NTFS image, which was created by using a random selection of files from the *GovDocs* corpus [32].

For benchmarking purposes and ease of analysis, we limit each container to a single CPU core. Further, we consider the processing functions one at a time in order to understand their intrinsic performance characteristics. The times shown are inclusive of all overhead, including network communications among the active containers.

Crypto hashing

Cryptographic hashing is a common forensic function. For this processing scenario, we selected the SHA-1 hashing algorithm. We developed a container that provides an RPC over TCP using the methods outlined in the previous section. The container returns the hash of the given data.

As the results in Table 4.1 indicate, as few as 12 containers practically saturate the available network bandwidth of 1GB/s.

Metadata extraction

The *ExifTool* [42] is commonly deployed and used to extract metadata from file content; it supports a large number of file formats and attributes. Listing 4.11 depicts the RPC wrapper code that – along with the *ExifTool* v10.10 executable is placed in a container image.

Considering the endpoints of the experimental space (Table 4.2 – 4 and 192 containers, respectively – we observe near-linear speedup from 5 to 192 MB/s. This is in line with expectations as the metadata extraction does not depend on I/O, and the workload is inherently data-parallel. Considering the whole range of parameters, we can see that the average throughput per container follows a bell curve distribution with an optimum return at around 32 containers.

Image Classification

Yahoo! recently released an open source image classifier, OpenNSFW (github.com/yahoo/open_nsfw). This is a deep neural network, built on top of Caffe (caffe.berkeleyvision.org), which

TABLE 4.2: *ExifTool* metadata extraction throughput vs. number of containers

Containers	4	8	32	64	96	192
MB/s	5.2	17	99	151	170	192
MB/s *per* cont.	1.3	2.1	3.1	2.4	1.8	1.0

TABLE 4.3: OpenNSFW classification throughput vs. number of containers

Containers	4	8	12	32	64	96	192
MB/s	0.4	1.4	2.5	3.8	7.2	10.9	21.3
Files/s	0.8	2.2	3.9	7.1	13.4	20.3	38.5

comes pre-trained to detect pornographic images. For every image that is processed, the system yields a value representing confidence in an image's resemblance to pornography.

This is a compelling case as it allows us to assess the cost of providing smarter tools for automated processing that provide results closer in abstraction level to that of the analyst. As Table 4.3 shows, these are expensive operations, and despite the linear scaling concerning the number of files classified, the absolute numbers are much lower than with other tools.

At the same time, compared to the current alternative of a human manually examining (thumbnails of) the images, even this unoptimized solution can classify 138,600 per hour or 3.33 million over 24 hours. By employing more modern CPUs, as well as GPUs, the system can be scaled up to the degree required to handle lab workloads.

Unlike previous examples, OpenNSFW is *already* provided as a docker container. We only needed to write a small wrapper function to encapsulate the embedded program as an RPC. As more developers adopt container solutions, such as Docker, integration with SCARF will become even easier.

Indexing Common Filetypes

Extraction of plaintext data from encoded documents is a basic step in forensic analysis. One of the more popular solutions is the *Apache Tika* (tika.apache.org) open-source project. It is specifically designed for this purpose and is often used in conjunction with an indexing engine such as *Solr*, or *ElasticSearch*.

From Table 4.4, , it is apparent that this workload is much more demanding and only scales sub-linearly. Particularly notable is the drop in processing rate per container between 96 and 192. Recall that there are only 96 physical cores, and it appears that, for this workload, the addition of hardware supported threads does not

TABLE 4.4: *Tika* text extraction vs. number of containers

Containers	4	12	24	48	96	192
MB/s	0.5	1.1	2.4	3.5	5.8	6.7
MB/s *per* cont.	.13	.09	.10	.07	.06	.03

TABLE 4.5: *Bulk extractor* throughput vs. number of containers

Containers	4	12	24	48	64	128
MB/s	3.5	17.9	22.7	54.8	59.4	151.5
MB/s *per* cont.	0.9	1.5	0.9	1.1	0.9	1.2

materially improve performance. This suggests that the workload is very effective at utilizing all the CPU's functional units; hence, the addition of threads only marginally improves on the amount of work being performed.

Bulk extractor

Bulk extractor [35] is a forensic tool used to analyze raw data streams. Using precompiled scanners based on *GNU flex*, it is a useful tool for extracting specific pieces of information, such as emails, URLs, IP addresses, and credit card numbers, from a data stream.

The process of wrapping and containerizing the tool was described earlier. What is interesting in this case is that *bulk_extractor* supports multi-threaded execution by default. Therefore, we approached this scenarios in a slightly different manner by exploring the optimal number of CPUs that a *bulk_extractor* container should have.

Interestingly, the best performance was achieved by limiting the tool to a single thread and a single CPU, using the saved resources to spawn additional containers. In other words, 48 one-core *bulk_extractor* instances work faster than one 48-core one. For this test case, we provided each container instance with 500MB of file data from the GovDocs corpus. [32].

Table 4.5 shows that the throughput per container remains relatively stable, and exhibits linear scalability. This is not a surprise as the workload is CPU-bound and data-parallel in nature.

Indexing filesystem metadata

Scalable Realtime Forensics utilizes ES for target metadata storage as well as storage of the results produced by tasks. For this benchmark, we extracted and parsed the NTFS metadata information of a 200-GB image containing approximately 620,000 files. Each NTFS-parsed record yields a JSON object of approximately 500 bytes.

TABLE 4.6: *ElasticSearch* throughput in standalone and cluster configurations

Containers	1	7
Records/s	1,299	14,236

Since ES is designed to run as a distributed service, it is crucial to consider different architectures as small changes could have substantial performance consequences. In this benchmark, we tested two architectures: a) single ES data node and b) production-style distribution containing seven nodes of various roles.

Table 4.6 illustrates that, indeed, moving from a standalone deployment to a small cluster yields super-linear improvement as synergies among the modules and fewer performance bottlenecks improve the effectiveness of the system. We did not test more extensive configurations (we would need more records); however, we expect larger configurations to scale-out well although the per-container performance may drop slowly as with most of the other services tested.

4.2.6 Metrics Summary

Inspection of a simple graphical plot, depicted in Figure 4.7, suggests that additional containers improve overall throughput. We see that both ExifTool and Bulk Extractor scale well, and while the deep neural-network-powered OpenNSFW indicates a lower rate of throughput per container, the throughput does increase. In contrast, Apache Tika throughput grows at a slower rate, suggesting a less scalable computation. The primary takeaway is that, with the SCARF architecture, we can add as many containers as the underlying hardware allows (the primary constraint is the number of CPUs). In other words, we can *easily* apply increasing amounts of hardware to combat the ever-increasing amount of volume present in a forensic investigation.

4.2.7 User experience

At this early stage, the user interface is fairly simple and consists of an interface with the ability to filter and drill down to the individual record. However, this is not sufficient for a production tool; we believe that there is substantial room for improvement in the user interface of forensic environments, and this subject deserves its own research effort and evaluation.

However, we do believe that integration of SCARF into our DSL will provide a seamless transition into a future-friendly interface for the end user – all analysts have to do is specify *what* needs to be done, and they do so using natural language. We now turn our attention to our proposed language, Nugget.

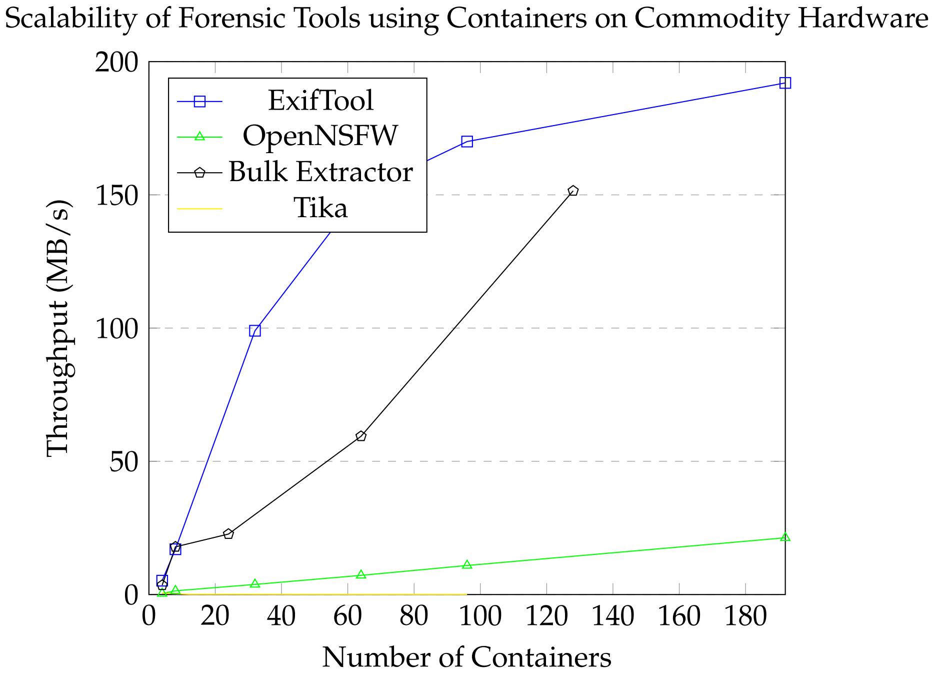


FIGURE 4.7: Scalability of Forensic Tools using Containers

4.3 Integration of Multiple Tools

As discussed in Chapter 3, the level of standardization and interoperability among cybersecurity products from different vendors, including open-source ones, is fairly low. Although understandable from a business perspective, this deficiency makes it difficult (and expensive) for customers to put together custom solutions and to have visibility across their entire IT infrastructure. It also hampers the adoption of custom data analytics and AI solutions and slows down the exchange of threat detection and mitigation solutions.

One promise of *Nugget* is to provide a common interface to the myriad of existing cybersecurity and DF tools. As a direct result, *Nugget* queries can operate across multiple tools, providing input to one tool from the results of another.

Here, we study the use of *Nugget* with two security tools. Specifically, we integrate Google's *GRR* incident response framework, and *Splunk*, the *de facto* standard for log aggregation. We demonstrate the utility of this type standardization to both tool developers and end-users analysts or IT administrators and discuss potential implications of having such a DSL becoming widely adopted across the entire domain of cybersecurity.

4.3.1 The Google rapid response (GRR) framework — distributed incident response

As discussed in Chapter 2, *GRR* is a framework specifically aimed at incident response [20]. It is an open-source, agent-based forensics tool aimed at providing live system data across multiple platforms. Furthermore, GRR's architecture consists of a one-to-many relationship between a server and the client computers running agents. When investigators wish to execute a forensic task, they initiate a *flow*. We offer brief descriptions of a server, a client agent, and flows below, with reference to Figure 4.8.

Server. The *GRR* server is primarily a Python and HTML based front-end, with SQL for backend database storage. It establishes a TCP listener for network communication with clients and workers and utilizes AES256 encryption for data transit.

The server also establishes multiple message queues that encompass various functions, such as worker coordination, CA enrollment, and interactive workers. The purpose of the message queuing functions is to support scalable forensic operations across enterprise-sized environments.

Client agents. These are initially distributed with enterprise software management mechanisms and are available for major platforms (OSX, Windows, and various types of Linux). Agents are responsible for executing flows on clients and responding to the server with results of flow execution. In addition, clients periodically transmit computer metadata (e.g., unique identifier, machine hostname, and machine metrics) to the server.

One notable feature of *GRR* clients is the builtin performance safeguards. Configurable mechanisms prevent the agent from adversely affecting enterprise operations, including CPU and memory limitations. Other notable mechanisms include transaction logging, heartbeat monitoring, and signed and encrypted network traffic.

Flows A flow is a unit of "forensic work" within GRR. It is responsible for executing one or more forensic operations on a client machine. This includes everything from file retrieval to running memory analysis with Rekall [9]. Flows are initiated by investigators and can be targeted at single or multiple clients simultaneously.

Flows are terminated after submitting results to the server. Results are serialized and stored as advanced forensics framework (AFF) objects [21].

Use case - file retrieval

One of the more common types of use cases for *GRR* involves a simple file hunt [61]. That is, the investigator simply wants to find a specific file on target workstations.

[9] https://github.com/google/rekall

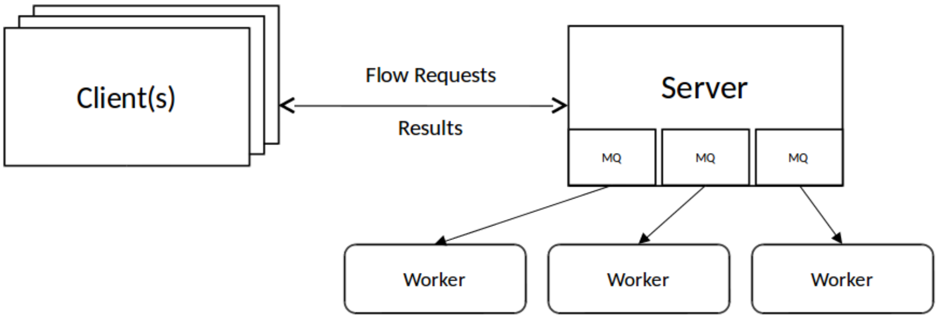


FIGURE 4.8: GRR Framework

To accomplish this in *GRR*, the investigator executes a series of point-and-click actions on the server's main GUI. These steps include initiating a new "find file" flow, providing runtime parameters (such as filenames to look for and hashes), selecting reporting type, and finally selecting which workstations flows should to execute the flow against.

Behind the scenes, *GRR* stores the request in the database and readies its execution by putting it on a processing queue. When the client checks in with the server, it will pull the new task and append it to its own execution queue. In this particular example, the client attempts to find the file, and if it exists, it will immediately return its hash value. The server checks against a local store to determine whether the file has previously been downloaded (by any client), and if not, it requests the file's binary data content. In addition, other checks are performed, such as a post-download hash comparison, to detect whether the file was changed during the flow execution.

4.3.2 Splunk

Splunk is the *de facto* standard for enterprise log monitoring. It natively supports several logging technologies, and with numerous third-party extensions (available through an integrated "app store"), it can catalog almost any type of log. We briefly discuss Splunk's architecture and a potential use case of utilizing Nugget to derive results from Splunk's API.

We briefly discuss Splunk's architecture and a potential use case of using Nugget to derive results from Splunk's API.

Splunk architecture

Various architectures are supported within Splunk, but primarily consist of a *forwarder*, an *indexer*, and a *search head*.

Forwarder. Forwarders are reasonably simple — they are configured to send specified logs to the indexer. These logs are usually structured in a known format from applications (e.g., Apache).

Indexer. The majority of log processing occurs at the indexer. It is responsible for indexing, time-stamping, and processing data sent from forwarders. Moreover, indexers support replication, data modification, and other desirable data operations.

Search head. The search head is the primary interface for the security analyst. It provides a GUI to initiate scans, it and provides mechanisms for scaling the searches in large-scale environments.

Use case - find hosts which visited a malicious domain

One traditional use case of Splunk is to filter logs based on the behavior of an endpoint's network traffic. In this example, investigators have learned that a compromised corporate endpoint has visited a malicious website. The investigators then turn to Splunk to determine which (if any) other endpoints have visited that same website.

This will traditionally be solved by having the investigator issue a Splunk query (using the Splunk-provided GUI). One potential query to retrieve this information could be (where field names are dependent on specific implementation): *index='proxy-weblogs' | url startswith="www.evildomain.com" | table requestor*. The results of this query are a table entries where the field 'requestor' (those endpoint requesting the domain in question) is given.

4.3.3 Integration using Nugget

One of the primary design goals of *Nugget* is ease of extensibility. In this use case, we extend *Nugget* to be compatible with both GRR and Splunk, allowing investigators to utilize the features of both security platforms *together*. This is accomplished via use of their individual APIs; however, it could also be integrated at a lower level if necessary (as we demonstrate with GRR).

GRR Integration

To illustrate the usefulness of *Nugget*, we began *GRR* integration with a inside-out approach: first, we added *Nugget* as an individual flow. Second, we integrated *Nugget* into the GRR GUI. Our final (and most realistic) use case integrated with GRR at the published API level, utilizing known constructs. This was an intentional repetition of work to showcase how any code architecture could support *Nugget*.

Adding Nugget to GRR. The most straightforward integration with *GRR* is to implement *Nugget* as a flow. It is the natural GRR approach to add a 'new' forensic tool (for example, it would be the natural approach for adding a tool such as 'scalpel'). While *Nugget* offers much more capability than just another forensic tool, this particular use case is interesting because it allows remote execution of *Nugget* queries directly on endpoints. Ultimately however, this use case showcases the extensibility of *GRR*, and not necessarily that of *Nugget* (and thus we limit our discussions).

Adding Nugget as a query within *GRR* GUI A more natural usage of *Nugget* within *GRR* is to use it as a 'driver' for all other forensic tasks. That is, we extended *GRR* to have a query entry field available on the main GUI landing page, allowing for the analyst to execute nugget queries across all registered clients. While this approach is a natural extension of both *Nugget* and *GRR*, it ultimately is not a canonical use of *Nugget* (and thus we limit our discussions).

Adding a GRR extension to Nugget The canonical extension of *Nugget* is to add *extractors* (see 4.1.2) to handle new data sources. In this case, we developed a *GRR* extractor specifically built for *GRR*'s REST API. Through the API, we can search clients, start flows, and generate reports, among other things.

To develop an extractor in *Nugget*, we created a JSON-based interface description and utilized *Nugget*'s included extension script. The script adds specified keywords (in this case, 'GRR') to its grammar, and then automatically regenerates its ANTLR-based parser. It next stubs out functions within the core *Nugget* codebase, along with typing information specified in the JSON description file. After the generation step, we code the compatibility layer to interact with GRR.

To interact with GRR, we utilized its provided HTTP-based API. Numerous libraries exist for Go (*Nugget* 's host language), which support such an API; therefore, building network communications with GRR is relatively easy. Here, we provide a brief description on adding 'File-Finder' capability into *Nugget*.

As discussed, the first step to extending *Nugget* is to add grammatical language constructs using the automated build tool. This is accomplished by constructing a JSON file from a template (see listing 4.12).

```
{
  "name": "file-finder",
  "codepath": "grr-filefinder.go",
  "operatesOn": ["strings"],
  "produces": ["strings","bytes"]
}
```

LISTING 4.12: Extending Nugget to support File Finder (GRR)

The build tool takes this file and generates a stubbed file containing the functions needed to fit within the language architecture. This consists of two parts: consuming an input and producing an output. For this example, the consumer is built to query a URL (the *GRR* server API), while the producer serializes the *GRR* response into an internal representation.

Once these functions are written (along with associated typing requirements), the functionality will be fully implemented within the *Nugget* runtime and can be used in conjunction with other (supported) forensic tools.

Adding a Splunk integration to *Nugget*

Splunk is integrated in much the same way as *GRR*- through an HTTP API. We create an interface description (similar to GRR's), and implement the generated stub methods by creating REST calls for specified user inputs.

In an near-identical manner to the *GRR* extension, *Splunk* is extended with the build-script listed in Listing 4.13.

```
{
  "name": "splunk-registry",
  "codepath": "splunk-registry.go",
  "operatesOn": ["strings"],
  "produces": ["strings","bytes"]
}
```

LISTING 4.13: Extending Nugget to support Splunk Registry

4.3.4 Experimental results: integrating Splunk and GRR

To showcase the power of integration, we utilize both *GRR* and *Splunk* in a single set of *Nugget* queries to conduct a realistic investigation of malware. However, before diving into the specific queries, we review the architecture of the tools' integration in Figure 4.9.

There are a few notable aspects of this architecture. First, the end user interacts *only* with the Nugget command line – no interaction with either Splunk or GRR (beyond the normal setup and deployment requirements for each product) is required. This interaction could be in the form of, inter alia, repeated tasks, automated executions, and AI or data analytic entry points; however, for our case, we focused on a single investigator. The second notable aspect of this architecture is the way in which *Nugget* can interact with both *GRR* and *Splunk* platforms. Finally, it is worth mentioning that in this scenario, *Splunk* is installed on servers, while *GRR* is installed on all endpoints. (Note: this is a simplistic view, and various components of *Nugget*, such as the backend database and resource manager, are left out for clarity).

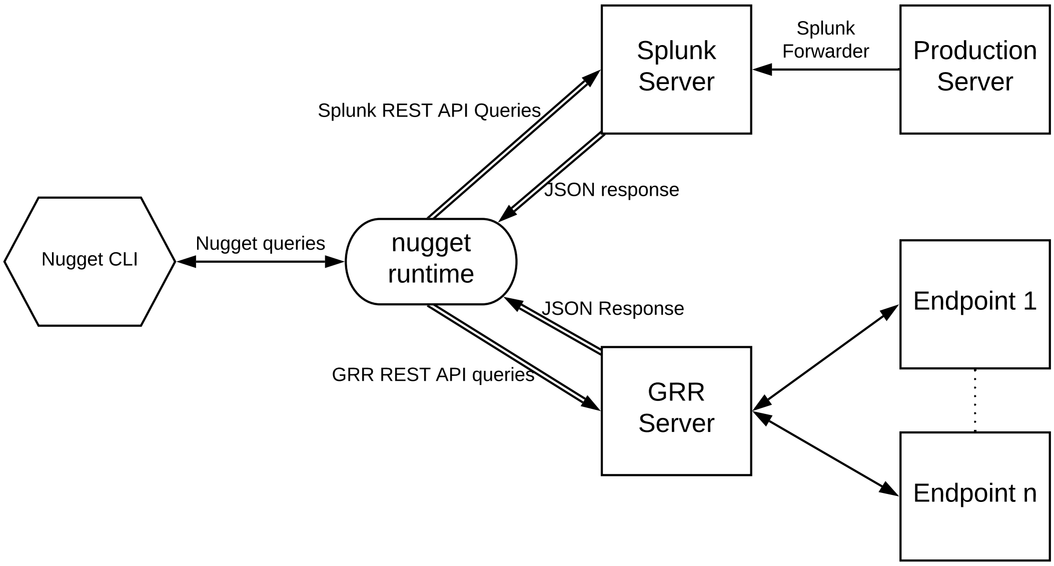


FIGURE 4.9: Integration Flow

Use case - indicator of compromise In this scenario, we simulated a sophisticated attacker gaining access to a server "victim" and installing a simple persistence module using the Microsoft registry. Once analysts notice questionable server behavior, they begin traditional incident response activities, which include searching the registry for undesirable "Run" and "RunOnce" entries. Using Splunk with a "registry module," they find a suspicious entry that loads binary data from the registry key "HKLM:Software/Microsoft/Windows/CurrentVersion/RunOnce" (the default behavior of a popular hacking toolkit, PowerShell Empire [10]). In an enterprise, the analysts would want to determine whether this indicator occurs anywhere else within their purview, and they would thus conduct an investigation using GRR. In our scenario, we utilized Splunk to extract indicators from a compromised server and GRR to search for those indicators across hundreds or thousands of computers.

Using the tools' respective GUIs could be cumbersome, and in this case (as with most) would involve manually handling with data extraction, data serialization, and data input into the specific tools (not to mention, learning each tool's individual GUI and respective nuances).

With *Nugget*, he could simply issue the natural query shown presented in Listing (4.14):

```
splunkextractor = net:192.168.100.27:8097 |  extract as splunk
bad_regkeys = splunkextractor | filter index="c2index; source="WinRegistry", key_path="HKLM\[..]\debug"
badregpath = bad_regkeys.key_path

grrextractor = url:192.168.100.12:8000 | extract as grr
active_grrclients = grrextractor | filter clients.checkintime > -10d
clientswithregistryIOC = active_grrclients | filter registrypath=badregistrypath

print clientswithregistryIOC.hostname, clientswithregistryIOC.IP
```

LISTING 4.14: Integrating *GRR* and *Splunk* via *Nugget*

LISTING 4.15: Expiremental Results of Splunk and GRR Integration

```
['winlab01, 192.168.2.51','winlab04, 192.168.2.54']
```

[10] https://www.powershellempire.com

While a primary design goal of *Nugget* is to be natural and understandable to the domain user (human investigator), we discuss the code to fully illustrate the power of *Nugget*.

On line 1, we utilize the Splunk extractors created (as described in section 4.1.2). Line 2 utilizes the *a priori* knowledge of a compromise in order to extract the specific indicators of the event according to the known filter; however, because *Nugget* has lazy execution, no computations are executed until this point, where the user will be shown the execution results.

On the nexst line (line 3), we extract and store the indicator of compromise (IOC) artifact for future use with *GRR*.

Line 5 establishes an extractor to utilize GRR data. Line 6 creates a variable containing a (future) list of all clients available to *GRR*. Line 7 filters that list to those clients which *match the indicators directly obtained from Splunk*! This is specifically where our integration efforts are realized.

Listing 4.15 showcases the results of the query execution (line 9), revealing the hostnames registered to *GRR* which match the indicators provided by *splunk*.

To summarize this (realistic) example use case, the investigator launched an incident response campaign (with *GRR*), based on input from a logging tool (*Splunk*). Importantly, our research allows the investigator to accomplish this using a *single interface with a single language*. This distinct difference from current techniques has significant implications, which we discussed in our introduction chapter and further analyze in our conclusions.

4.4 Conclusion

Final Integration. The life of a *Nugget* query within an integrated processing runtime is depicted in Figure 4.10. For this example, we showcase a container based runtime, such as with *SCARF*.

A *Nugget* query can be created on the command line or with a series of saved commands, known as scripts (saved within a *.nug* file). Alternatively, the query could be created with a user-based GUI, which is a subject for future development.

The query then enters the parsing engine. The parsing engine sets the stage for query execution by taking care of input validation and memory allocations, and provides instant feedback to the user in the case of an erroneous statement. The engine also constructs filters based on user input — importantly, filtering occurs *prior* to data loads, allowing for end-users to prioritize work in large datasets.

When statement evaluation is required (recall, *Nugget* is a delayed evaluation language), the engine will issue a data-load command to the data broker. The data-load only occurs the first time the data-load is needed for the specified target (multiple subsequent data-load commands are ignored). For optimal performance the target data should be local to the data broker; however, network-based acquisition is also possible. The data broker, in turn, distributes the target dataset to all available consumers.

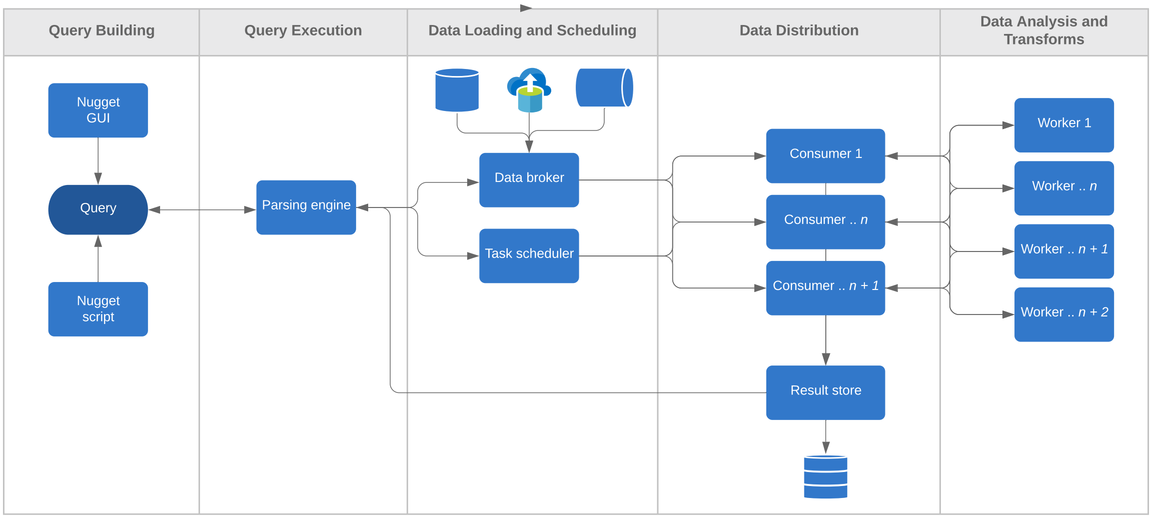


FIGURE 4.10: *Nugget* query flow in a container-based architecture

In our research, we built loaders (or, 'consumers') to handle the following:

- *ntfs*
- *ext3/ext4*
- *pcap*
- *RAM*
- *grr*
- *splunk*

The task scheduler will receive all subsequent commands, such as 'execute md5', and route them to the appropriate consumers based on how the data was initially distributed.

For clarity, this diagram separates the data broker from task scheduling; in our technical implementation, they coexist. However, the point is that all tasks that distribute to a cluster are tracked (in a message queue), and remain until a successful completion acknowledgement is received.

Once the data is distributed, the user can begin executing analysis against any type of supported worker. In our research, we built workers for the following:

- *md5*
- *sha1*
- *sha256*
- *pslist*
- *grep*
- *tika*
- *bulk_extractor*
- *open-nsfw*
- *exiftool*

Finally, results from workers are fed into a results cluster. Subsequent queries relying on results from prior computations are either retrieved from the parsing engine's cache (if small enough), or are retrieved from the storage repository. A key idea to this feature is allowing 3rd party users (or applications) access to the results repository, which can potentially eliminate the need for re-execution of *Nugget* queries across a large runtime.

Closing thoughts. In this chapter, we discussed the implementation details of our final solution. We then covered detailed results from use cases generated and outlined in our methods chapter. We showcased a scalable, distributed solution that easily incorporates any digital forensic tool. Furthermore, we demonstrated that the *Nugget* DSL is indeed viable and that extending and incorporating third-party tools is easy.

In the next chapter, we detail these points, finalize our thoughts, and consider steps for future research.

Chapter 5

Conclusions

This research aimed to identify and implement a robust framework to address myriad issues across the domain of DF, particularly a lack of standardization amongst researchers and law enforcement analysts. The solution presented in this thesis demonstrates that the proposed *Nugget* language offers a robust foundation upon which researchers, analysts, and law-enforcement can build future tools and investigations.

The rest of this chapter briefly summarizes these issues, our solution, and contributions. It then discusses future research and recommendations.

5.1 Recapitulation of Issues

To finalize the discussion of issues facing DF, we briefly consider its history. In the early 1980s, relatively few types of computing platforms existed, and relatively few people were using them. The first digital forensic tools were organically developed to answer questions that contemporary investigators faced. However, as technology and its usage grew exponentially, little to no standardization of tools existed to assist investigators in studies of cyber criminals.

If we fast-forward to today, where billions of users are utilizing thousands of different computing technologies, a survey of DF tools suggests that not much has fundamentally changed. Various forensic tools address *specific* investigator questions, and although some toolkits exist that address a large subset of needs, no comprehensive standardization has been adopted.

The lack of such adoption has had several consequences for DF and the greater cyber-security community. Foremost, analysts do not have a way in which to explain their investigative actions in a clear, repeatable, concisely and computationally specific format; this has clear implications in a court of law: if a subject matter expert cannot readily verify an analyst's research, then the resulting evidence loses veracity.

Without a descriptive language, educators have no choice but to provide educational material that revolves around tools rather than teaching a universally applicable, vendor agnostic forensic computation.

Another consequence is little to no tool interoperability across the domain of cybersecurity tools. As discussed, hundreds of popular tools exist, each fulfilling a specific function. Indeed, analysts utilize whole sets of tools during the course of an investigation. However, without interoperability, analysts are forced to transfer data between them manually. This is time-intensive and error-prone, and it is unsustainable when considering the scale of modern DF investigations.

A similar consequence is that tools are not readily compatible with modern high-performance processing frameworks. They cannot be "woven into the computational fabric" of big-data systems such as Hadoop, Spark, and NoSQL. Similarly, machine learning and AI cannot be readily applied to the datasets of forensic investigations (datasets that would be ideal candidates for automated, high-level processing). As the average digital forensic investigation faces ever-increasing amounts of data, these high-performance processing engines are becoming increasingly vital.

Tool development has suffered from a lack of standardization. Investigators, or end users, and tool developers have no means of specifying *what* they need to conduct investigations. As a result, vendors often provide unnecessary, incomplete, or inexact features.

Finally, there is a lack of tool *verification* for forensic operations. That is, the following question remains: how can analysts trust that a task, such as finding files, has completed *fully*, with no false positives or false negatives? In the case of black-box tools, investigators, who themselves have no established oversight for verification, are forced to trust black-box solutions. In the case of open-source tools, analysts are commonly unqualified to conduct robust code reviews. The lack of tool verification is a particularly important concern because of the rapid growth, mutation, and adoption of underlying computing technologies (e.g., the following question could be asked: if an operating system is updated, is the forensic tool still trustworthy?).

In short, DF has foundational issues that cause myriad problems downstream. In our research, we provide a solution *at the foundation layer*, thereby immediately solving these problems.

5.2 Summary of Research

Our preliminary research into related work revealed several proposed solutions to issues described in Chapter 1. These propositions range from highly specific mathematical models to overly abstract ones, and none of them addresses each issue we identified. Indeed, it is important to note that most related research focuses on singular "downstream" problems. In other words, while other subject matter experts have witnessed the symptoms, few have identified the disease, and none have offered practical solutions.

We divided the research into two broad phases. First, we developed a DF runtime "engine." Second, we built a DSL for DF that can describe forensic computations unambiguously and run them with the engine developed in the first step. We discuss the latter first.

5.2.1 The language

Requirements. While the *SCARF* engine provided the execution runtime, the primary contribution of our research is the DSL we designed and implemented. Domain-specific languages are programming languages of limited expressiveness, built for a specific purpose and for the benefit of a specific group of users. Our DSL, named Nugget, provides investigators and analysts with the syntax and semantics required to express discrete digital forensic operations. During *Nugget* 's design phase, we identified several core requirements:

- Formal — As with any programming language, statements must resolve into explicit computations.
- Natural — Statements must be natural to the subject matter expert who is expected to use the DSL.
- Delayed execution — Execution of forensic computation should not occur until results are either required for additional computation or presented to the user.
- Tool agnostic — Syntax should be “open,” allowing for any tool to be added to the language.
- Extensible — The DF community must be able to integrate their own tools into the language.

A selective discussion about our identified requirements is warranted. First, the language must be natural to the end user. In our case, the primary end user is law enforcement, who will likely have little to no computer science or programming background. We also wish to consider students of DF, academic researchers, and non-human interfaces (e.g., to support future APIs for third-party tool verification).

A desirable aspect of query languages is delayed (or deferred) execution. By delaying the execution, investigators can build complex queries involving numerous dependent variables, without having to wait for individual statement execution until computationally necessary. This is beneficial from a usability point of view as well as for performance reasons, as the language can avoid execution of unnecessary computations.

Given the sheer volume of tools built for DF, the language must be tool-agnostic. That is, syntax and semantics cannot be tailored to any existing tool. Indeed, in the DF community, many researchers and analysts have favorite tools or methodologies for solving specific tasks (usually because of requirements, although preference is also a driver of tool usage). Since *Nugget* is built as an open framework, no specific tool or process should be baked into the core processing fabric; instead, each tool integration should be treated as an extension of the core language. This

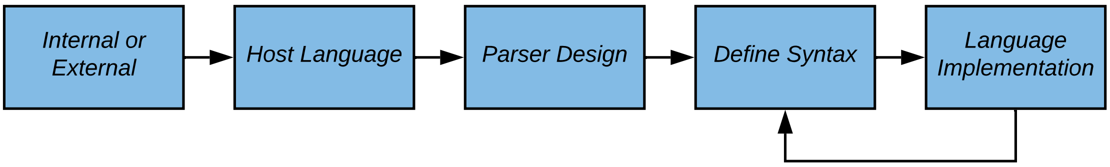


FIGURE 5.1: Summary of DSL development process

also allows for "drag and drop" replacement of "competing" tools, for example if a user has a requirement for one tool over another.

A desirable aspect of any framework is its ease of extensibility. In this case, users (such as non-technical law enforcement analysts) must have the ability to incorporate their favorite (or required) tools into the *Nugget* processing framework. While we provided a few tool integrations, they represent a relatively small subset of the hundreds of existing digital forensic tools.

Implementation details - ***Nugget*** *Nugget* was systematically implemented to meet our established requirement criteria, summarized above. Here, we briefly outline the implementation process, including reflections on design choices, with a reference at Figure 5.1.

In our preliminary research, we sketched out desirable syntax and semantics of the language. We essentially asked, "What are the common actions or tasks in a digital forensic investigation?". From prior experience in the realm of DF, we were able to a) select verbiage to match the natural phrases involved and b) reference a common task in Listing 5.1. Expressions such as “extracting” from a target hard drive, “filtering,” and “printing” are all-natural to our target users. In comparison, sub-fields, such as “content,” can be more opaque; however, future iterations of *Nugget* will have development environment constructs to assist end users.

```
files = file:target.raw | extract as ntfs[63,512]
pdf_files = files | filter filename == "*pdf"
hashes = pdf_files.content | sha1, md5
print hashes
```

LISTING 5.1: PDF file extraction, filtering (by size), and hashing

With an idea of preliminary syntax constraints, our next consideration of DSL development was an *internal* vs. *external* language. Internal languages are extensions of general-purpose programming languages, such as Scala (which incidentally has dedicated DSL constructs). While these languages are extremely powerful because they inherit host-language constructs, the DSL is chained to host language semantics. In other words, users who wish to extend or improve the DSL must learn the host language.

For *Nugget*, we recognized a need for simple syntax and semantics to ease the learning curve for potentially non-technical end-users. This naturally led us to select an **external** DSL approach. In an external DSL, we have the freedom to develop the symbols, syntax, and semantics used in the language, at the cost of losing host-language constructs. As a simplistic example, this means users cannot perform common programming computations (such as string manipulation) in *Nugget* without directly adding that feature to the DSL. However, we believe that because our target users are likely to have no programming background, the need to have natural, simplistic and expressive syntax trumps baked-in integration of host language functionality.

The next critical decision was choosing the host implementation language. After a brief trial-and-error phase using Scala, we opted for Google's Golang as the implementation language for *Nugget*. Golang is an ideal choice because of the intersection of three desirable traits: widely available support, relative ease of use, and high performance. In addition, it is growing in popularity in international rankings, and we expect that non-technical *Nugget* users will be able to find help. At the technical level, however, Golang has a significant shortcoming in the lack of support for generics. With *Nugget*, this was addressed with extensive use of "empty" interfaces. Future implementations of Golang will add support for generics, allowing for some simplification of *Nugget*'s codebase [69].

The next decision to make was how to parse user input. While we initially decided to create a parser by hand, we quickly realized that the utilization of a parser generator provides robust capability at little cost. We subsequently decided to utilize the open-source, widely used *ANTLR* parser generator for *Nugget*. As with other generators, it requires DSL developers to create a language description in BNF format and will generate host-language constructs that parse user input. Additional features of *ANTLR* include error-handling, logging, and support for several output languages. Finally, it is beneficial to use a generator when the language is quickly evolving. However, utilization of a third-party tool incurs a cost – any future additions or extensions to the language will require modifications to the language description and regeneration of the parser. To minimize the impact, we provide a build script that handles all *ANTLR* interactions, hiding their complexities from the non-technical end user.

By this point, we knew to use *EBNF* formatting to define *Nugget* syntax and semantics. Thus, we were able to formally define the *Nugget* language.

At this stage, we began implementing the language; that is, we had to program the actual *Nugget* behavior. Several aspects of this process are notable.

First, the programming goal was to establish the core framework required to execute the input syntax given in 5.1. Having the framework, we then had to decide on a set of initial tools to integrate and that could showcase the power of *Nugget* while also providing a robust set of core functionality. For this, we considered three common use domains within DF: network forensics, hard drive forensics, and memory forensics. For each, we selected a prominent open-source project to integrate, resulting in *libpcap*, *The Sleuth Kit*, and *Volatility*. This core functionality allowed us to showcase a token investigation into the M57 dataset, as described in detail in

Chapter 4 as well as in our publication with DFRWS [89]. Finally, robust logging capabilities were developed that allow for postmortem analysis of the investigation itself, should the need arise.

Throughout the implementation process, we continuously conducted testing. Testing feedback allowed us to discover shortcomings and implement syntax improvements throughout development.

Extending and Showcasing *Nugget*. Once the *Nugget* framework was established with several DF tools, we showcased how it can be used to address a significant shortcoming in the cyber-security community, namely, tool interoperability, which is the ability to feed results from one tool directly into another tool. As detailed in a paper published with a 2019 ARES conference [90], we demonstrated how two popular security tools can seamlessly integrate with *Nugget*. Specifically, we built a token incident-response case where analysts feed indicators of compromise from *Splunk* into Google's agent-based IR tool, *GRR*, with the goal of generating a listing of all infected devices. If this was a real investigation, analysts would manually need to utilize both tools (likely with cumbersome GUIs) and manually copy and paste data from one tool to another. This leaves no audit trail, is prone to user errors, and is time-consuming. Instead, the simple *Nugget* query is given in Listing (5.2):

```
 1 splunkextractor = net:192.168.100.27:8097 |  extract as
      splunk
 2 bad_regkeys = splunkextractor |
 3     filter index="c2index; source="WinRegistry",
 4     key_path="HKLM\[..]\debug"
 5 badregpath = bad_regkeys.key_path
 6
 7 grrextractor = url:192.168.100.12:8000 | extract as grr
 8 active_grrclients = grrextractor |
 9     filter clients.checkintime > -10d
10 clientswithregistryIOC = active_grrclients |
11    filter registrypath=badregistrypath
12
13 print clientswithregistryIOC.hostname
```

LISTING 5.2: Integrating *GRR* and *Splunk* via *Nugget*

LISTING 5.3: Expiremental Results of Integration

```
['winlab01','winlab04']
```

The integration specifically occurs on lines 11 and 12, where a *GRR* filter is generated using registry information from *Splunk*.

Lessons learned Reflecting on our development process, we could have made several improvements. Most importantly: as we tested the development versions of the language, we continuously modified the core *Nugget* syntax. This led to confusion throughout the development process, resulting in a loss of time. For future DSL developments, we recommend thoroughly defining the DSL syntax prior to writing the first line of code.

Second, we lost time when integrating *Nugget* with *SCARF*. Instead of integration with the first development versions of *Nugget*, we fully developed *Nugget* with "stand-in" runtimes. While replacing the backend runtime with *SCARF* was ultimately successful, it was time-consuming.

Finally, we recommend building a custom input parser and interpreter from scratch. While we had success using *ANTLR*, it is ultimately reliance on a third-party library. Reliance on any external libraries is not ideal, and in this case, it is only sporadically updated. In addition, that reliance complicates language extensibility, especially for non-technical users.

5.2.2 The scalable runtime engine

Requirements. Our engine had several requirements:

- Scalable — the solution must be able to handle arbitrarily large datasets (such as those faced in future enterprise-sized investigations) with the simple addition of computing power.
- Input agnostic — the solution must be able to ingest data from arbitrary sources, such as traditional hard disks, SSDs, and 100 Gigabit networks.
- Extensible — the solution must be able to easily incorporate additional functionality.
- Distributed Pooling - the solution must pool available resources and distribute tasks accordingly.
- Logging — the solution must provide robust logging mechanisms to detect and recover from task failures inherent in scalable solutions.

Implementation details — *SCARF* Our solution, as outlined in Chapter 4, is *SCARF*, and it is comprised of four major parts. First, a data broker is responsible for ingesting data into an internal format. Second, a *task manager* takes input, which consists of forensic tasks (such as "hash all files for a given input source"), from an end user and coordinates execution against a pool of workers. Third, a *worker pool* executes jobs as they are received, feeding results back to the task manager. Fourth, the task manager feeds results into the *results repository*. This simplified explanation is illustrated in 5.2.

While we limit an in-depth discussion of *SCARF* and refer readers back to our details in Chapter 4, there are a couple of notable aspects of its *implementation* worth revisiting. First, the basic unit of computation is a *worker*, and workers are grouped into a pool. Each worker is an individual *container*, which is built as a small runtime

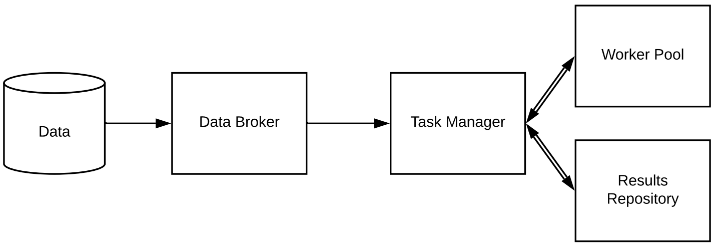


FIGURE 5.2: Abbreviated diagram of SCARF.

specifically for a single type of forensic task. For example, our initial implementation supported the following types of containers: *SHA1, grep, Apache Tika, Yahoo open_nsfw, bulk_extractor, and ExifTool*. The workers are unaware of the greater 'computing fabric' they are a part of, which simplifies the scaling process.

The scaling process is the second notable aspect of this architecture. It is handled by an orchestration suite, allowing the user granular control over the number of each type of individual worker deployed. For instance, if investigators identify a high-priority forensic task, they can easily increase the number of *grep* containers at the expense of reducing the number of *SHA1* containers deployed. In our implementations, we utilized *Docker* for both containers and their orchestration; however, it could easily be replaced by *Kubernetes*, *rkt*, or others.

The final discussion point on implementation is how we handle the storage of results. As previously stated, analysts are faced with exponentially increasing amounts of data, and compiled results from forensic operations are consequently ever-increasing. To solve this challenge, we implemented a scalable datastore for our results repository, allowing for results to be retrieved in near real-time. We specifically utilized *ElasticSearch* (ES). This type of scalable architecture allows investigators to add additional hardware as computational demands, such as capacity or processing speed, grow over time.

While we provide a summary the details regarding the engine here, further information can be found in Chapter 4, as well as our publication with DFRWS [88].

5.3 Summary of Contributions

In summary, this research has yielded eight distinct contributions to the field of digital forensics, cyber-security, and information assurance.

1. Provide digital forensic investigators the means to specify a digital forensics computation that is both practical and formal;
2. Provide digital forensic investigators a means for reproducing digital forensic investigations;
3. Provide an abstract layer of communication suitable for use between forensic analysts, law enforcement, and tool developers;

4. Perform a mechanism by which digital forensic tools can be benchmarked;
5. Provide external entities (e.g., NIST) a means to validate digital forensics tools;
6. Provide educators a tool-agnostic medium to teach digital forensics processes;
7. Provide a framework enabling interoperability between digital forensic tools;
8. Create a flexible, scalable, and container-based runtime to demonstrate effective usage of *Nugget* queries.

5.4 Looking Ahead

Looking ahead, we stress the potential impact of our research and identify improvements that will increase our overall impact.

5.4.1 Research impacts

As a result of this research, we expect and hope for several community-wide impacts.

First, we expect governing agencies, such as the National Institute for Science and Technology (NIST), to adapt our approach in the verification of forensic tools used in legal proceedings. Logistically, the only new requirement by tool vendors would be the integration of an API allowing for *Nugget* query execution (as demonstrated with both *grr* and *splunk*). Then, NIST could run quality assurance checks using the vendor's tools against a known baseline.

Second, we hope for the widespread adoption of Nugget throughout the analyst community. If a community grows around the tool, it will standardize all forensic operations – similarly to what SQL did for the database community before its invention. In addition, add-ons and expansions to Nugget could be shared amongst the community, thereby removing the potential overhead of specific tool integration.

Third, we expect *Nugget* to improve court proceedings involving DF. As there is no current method to reproduce a digital forensic investigation, there is no scientific method exists to verify a law enforcement analyst's conclusion. Similarly, the government verification of tools naturally lends itself to simplification of legal proceedings.

5.4.2 Future improvements

We have identified several future improvements for our research that will drastically increase chances of widespread adoption.

The first item on our short-list is a full-featured integrated development environment (IDE) for *Nugget*. This drastically decreases its learning curve and would be an ideal platform for future GUI research.

Second, additional *Nugget* extensions, such as support for mobile filesystem extractors (iOS and Android) or AI-based transforms (such as TensorFlow hooks) should be created.

Finally, we wish to implement a custom *Nugget* syntax parser for user input. While our ANTLR-based solution is robust and reasonably extensible, reliance on a third-party for a critical component is far from ideal ideal.

5.5 Final Thoughts

In closing, our framework provides a comprehensive foundation for DF that in turn provides a solution to a wide range of issues in the DF and cyber-security communities. First, *SCARF* offers a framework to integrate arbitrary tools across a distributed processing fabric. Second, *Nugget* provides a computationally specific, yet human-readable language to describe and execute forensic operations.

Together, these frameworks provide a scalable platform for analysts to execute *scientifically sound* digital investigations.

We sincerely hope all types of analysts and researchers, from law-enforcement to academia, will adopt our digital forensics language.

Appendix A

Code Listings

A.1 Nugget Grammar

```
grammar Nugget;

@header {
    // import "../NTypes"
}

prog: (         define_assign |
       operation_on_singleton |
                singleton_var )*
       EOF
;

define_assign:   define |
                 define_tuple |
                 assign
;

define: ID nugget_type LISTOP? ;

define_tuple: ID ’tuple[’ (’,’? nugget_type)+ ’]’ LISTOP?;


assign: ID ’=’ STRING (’|’ nugget_action)* |
        ID ’=’ ID     (’|’ nugget_action)*
;

operation_on_singleton: singleton_op  ID (’,’ ID)* output_as?;

output_as: ’as’ output_type;
output_type: ’json’;

```

```
singleton_op: ('type' | 'print' | 'size' | 'typex' | 'printx' | 'raw') ;

singleton_var: ID;

nugget_type:
      'string'     |
      'sha1'       |
      'md5'        |
      'ntfs'       |
      'file'       |
      'packet'     |
      'pcap'       |
      'exifinfo'   |
      'datetime'   |
      'memory'     |
      'http'       |
      'listof-md5' |
      'listof-sha1'|
      'listof-sha256'
      ;


nugget_action: action_word ;

action_word:
    filter            |
    'extract' asType  |
    'sort'    byField |
    'sha1'            |
    'md5'             |
  'sha256'          |
  'getGetRequests'  |
  'diskinfo'        |
  'union'    ID     |
  'pslist'          |
  '%%%'
;

asType: 'as' nugget_type (byteOffsetSize)?;
byField:'by' ID;

byteOffsetSize : '['INT ',' INT ']';

filter : 'filter' filter_term (',' filter_term)*;
filter_term: ID COMPOP STRING;
```

```

COMPOP: (’>’ | ’<’ | ’>=’ | ’<=’ | ’==’);
LISTOP: ’[]’;

INT : [0-9]+;
ID : [a-zA-Z]+ (’.’ [a-zA-Z]+)?;
STRING: ’"’ (’""’|~’"’)* ’"’;

WS : [ \t\r\n]+ -> skip;
NL : ’\r’? ’\n’;

LINE_COMMENT: ’//’ ~[\r\n]* -> skip;
```

A.2 Sample Code - *DEX*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DEX_root>
<DEXroot version="0.0">
  <CreationDate>2009-03-16 22:26:25</CreationDate>
  <DiskImage MD5Sum="6aa7dd7aa21061cca79e1d85cc1a8450">
    <Filename>disk-image</Filename>
  </DiskImage>
  <PartitionTable
ParentPtr="/DEXroot/DiskImage[@MD5Sum=6aa7dd7aa21061cca79e1d85cc1
a8450]">
    <SectorSize>512</SectorSize>
    <Offset>0</Offset>
    <Version>Darwin 9.6.0 Darwin Kernel Version 9.6.0: Mon Nov 24
17:37:00 PST 2008; root:xnu-1228.9.59~1/RELEASE_I386
i386</Version>
    <CommandLine>fdisk -d disk-image</CommandLine>
    <Volume>
      <StartSector>63</StartSector>
      <EndSector>32129</EndSector>
      <Description />
      <Type>7</Type>
    </Volume>
  </PartitionTable>
  <MasterFileTable
ParentPtr="/DEXroot/PartitionTable/Volume[StartSector=63]">
    <Version>The Sleuth Kit ver 3.0.1</Version>
    <CommandLine>istat -o 63 disk-image 27</CommandLine>
    <entryAddress address="27"
MD5sum="f7c3c7307accce6ef2797d7beb9ecd7e">
<MFTEntryHeader>
<Entry>27</Entry>
<Sequence>1</Sequence>
<LogFileSequenceNumber>0</LogFileSequenceNumber>
        <Links>1</Links>
</MFTEntryHeader>
<STANDARD_INFORMATIONAttribute>
<Flags />
<OwnerID>0</OwnerID>
<Created>Wed Mar 11 09:38:28 2009</Created>
<FileModified>Wed Mar 11 09:38:28 2009</FileModified>
<MFTModified>Wed Mar 11 09:38:28 2009</MFTModified>
<Accessed>Wed Mar 11 09:38:28 2009</Accessed>
</STANDARD_INFORMATIONAttribute>
<FILE_NAMEAttribute>
<Flags />
<Name>finn.jpg</Name>
<ParentMFTEntry>5</ParentMFTEntry>
<Sequence>5</Sequence>
<AllocatedSize>0</AllocatedSize>
        <ActualSize>0</ActualSize>
        <Created>Wed Mar 11 09:38:28 2009</Created>
        <FileModified>Wed Mar 11 09:38:28 2009</FileModified>
        <MFTModified>Wed Mar 11 09:38:28 2009</MFTModified>
        <Accessed>Wed Mar 11 09:38:28 2009</Accessed>
      </FILE_NAMEAttribute>
      <Att>
        <STANDARD_INFORMATION Resident="Resident" />
        <FILE_NAME Resident="Resident" />
        <SECURITY_DESCRIPTOR Resident="Resident" />
        <DATA Resident="Non-Resident">2517 2518 2519 2520 2521 [ ... ]</DATA>
      </Att>
    </entryAddress>
  </MasterFileTable>
  <File
ParentPtr="/DEXroot/MasterFileTable/entryAddress[@address=27]"
MD5Sum="7a35e1fb89cd05b8465d97f6b402404c">
    <Version>The Sleuth Kit ver 3.0.1</Version>
    <CommandLine>icat -o 63 disk-image 27</CommandLine>
    <Filename>finn.jpg</Filename>
  </File>
  <Exif
ParentPtr="/DEXroot/File[@MD5Sum=7
    a35e1fb89cd05b8465d97f6b402404
c
]">
    <Version>Jhead version: 2.87   Compiled: Mar 11
2009</Version>
    <CommandLine>jhead finn.jpg</CommandLine>
    <CameraMake>Canon</CameraMake>
    <CameraModel>Canon PowerShot A720 IS</CameraModel>
    <DateTime>2008:10:19 09:38:59</DateTime>
  </Exif>
</DEXroot>
```

A.3 SWGDE Best Practices for Image Authentication

```
7.1 The original imagery shall be preserved. Any processing should be
   applied only to a working copy of the imagery
7.2 Assess the image structure to determine whether factors are present
   that can answer the examination request.
Image structure examinations may include, but are not limited to:
    7.2.1 An examination of the file format of the imagery
    7.2.2 An examination of the metadata of the imagery
        Metadata may be useful in identifying the source and processing
   history of the file, but can be limited, absent, or altered.
        Metadata may include:
            Camera make/model/serial number
            Date/time of creation or alteration
            Camera settings
            Resolution and image size
            Global Positioning System (GPS) coordinates/elevation
            Processing/image history
            Original file name
            Lens or flash information
            Frame rate
            Thumbnail information
    7.2.3 An examination of the imagery file packaging (container
   analysis). This analysis may include but is not limited to:
        Hex level header, footer, or other information about the file
        Exchangeable image file format (EXIF) information
7.3 Image Content
    7.3.1 Artifact features
        Chromatic aberrations
        Breaks in compression blocking or patterns
    7.3.2 Physical aspects of the scene
        Lighting, contrast
        Scale
        Composition
        Physics
        Temporal or geographic inconsistencies
    7.3.3 Human characteristics
        Hair detail
        Scars, bruises, or blemishes
        Creases
        Vein patterns
        Skin contact
        Movement
    7.3.4 Evidence of staging
    7.3.5 Photographic conditions
        Focus
        Depth of field
        Sharpness/blur
        Perspective
        Grain structure
        Noise
        Lens distortion
```

A.4 Sample Code - *DERRIC*

```
format JPG

unit byte
size 1
sign false
type integer
endian big
strings ascii

sequence SOI APP0JFIF APP0JFXX? not(SOI,
APP0JFIF, APP0JFXX, EOI)* EOI

structures
SOI { marker: 0xFF, 0xD8; }

APP0JFIF {
    marker: 0xFF, 0xE0;
    length: lengthOf(rgb) + (offset(rgb) -
    offset(identifier)) size 2;
    identifier: "JFIF", 0;
    version: expected 1, 2;
    units: 0 | 1 | 2;
    xdensity: size 2;
    ydensity: size 2;
    xthumbnail: size 1;
    ythumbnail: size 1;
    rgb: size xthumbnail * ythumbnail * 3;
}

DHT {
    marker: 0xFF, 0xC4;
    length: size 2;
    data: size length - lengthOf(marker);
}

SOS = DHT {
    marker: 0xFF, 0xDA;
    compressedData: unknown
    terminatedBefore 0xFF, !0x00;
}
```

Bibliography

[1] 18 US Code § 1030 - Fraud and related activity in connection with computers. https://www.law.cornell.edu/uscode/text/18/1030.

[2] Alink, W., Bhoedjang, R., Boncz, P., and de Vries, A. Xiraf – xml-based indexing and querying for digital forensics. *Digital Investigation 3* (2006), 50 – 58. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

[3] Ayers, D. A second generation computer forensic analysis system. *Digital Investigation 6* (2009), S34 – S42. The Proceedings of the Ninth Annual DFRWS Conference.

[4] Backus, J. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *Proceedings of the International Conference on Information Processing* (1959), pp. 125–132.

[5] Bos, J. V. D., and Storm, T. V. D. Bringing domain-specific languages to digital forensics. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ICSE, pp. 671–680. 10.1145/1985793.1985887.

[6] Brian Carrier. Open Source Digital Forensics Tools, 2003. http://www.digital-evidence.org/papers/opensrc_legal.pdf.

[7] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. *ACM Queue 14* (2016), 70–93. http://queue.acm.org/detail.cfm?id=2898444.

[8] Carrier, B. *The Sleuthkit*. http://www.sleuthkit.org/sleuthkit/.

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

[10] Case, A., Cristina, A., Marziale, L., Richard, G. G., and Roussev, V. Face: Automated digital evidence discovery and correlation. *Digital Investigation 5* (2008), S65 – S75. The Proceedings of the Eighth Annual DFRWS Conference.

[11] Casey, E. What constitutes a proper education? *Digital Investigation 11*, 2 (2014), 79 – 80.

[12] Casey, E., Back, G., and Barnum, S. Leveraging cybox to standardize representation and exchange of digital forensic information. *Digital Investigation 12* (2015), S102 – S110. DFRWS 2015 Europe.

[13] Casey, E., Biasiotti, M. A., and Turchi, F. Using standardization and ontology to enhance data protection and intelligent analysis of electronic evidence. University of Lausanne.

[14] Castle, G. Find all the badness collect all the things. Blackhat 2014, 2014.

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

[16] Chamberlin, D. Encyclopedia of database systems, 2009.

[17] Chamberlin, D. D., and Boyce, R. F. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (New York, NY, USA, 1974), SIGFIDET '74, ACM, pp. 249–264.

[18] Codd, E. F. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (New York, NY, USA, 1971), SIGFIDET '71, ACM, pp. 35–68.

[19] Codd, E. F. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[20] Cohen, M., Bilby, D., and Caronni, G. Distributed forensics and incident response in the enterprise. *Digital Investigation 8* (2011), S101 – S110. The Proceedings of the Eleventh Annual DFRWS Conference.

[21] Cohen, M., Garfinkel, S., and Schatz, B. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. *Digital Investigation 6* (2009), S57 – S68. The Proceedings of the Ninth Annual DFRWS Conference.

[22] Corbet, J. Notes from a container, 2007. https://lwn.net/Articles/256389/.

[23] Forensics knowledge area, 2019. https://www.cybok.org/.

[24] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (Jan. 2008), 107–113. 10.1145/1327452.1327492.

[25] Ekelhart, A., Kiesling, E., and Kurniawan, K. Taming the logs - vocabularies for semantic security analysis. *Procedia Computer Science 137* (2018), 109 – 119. Proceedings of the 14th International Conference on Semantic Systems 10th – 13th of September 2018 Vienna, Austria.

[26] Esienberg, T. The cornell commission: On morris and the worm. *Communications of the ACM* (Jun 1989).

[27] FBI. The morris worm, Nov 2018.

[28] A road map for digital forensic research, 2001. https://www.dfrws.org/sites/default/files/session-files/a_road_map_for_digital_forensic_research.pdf.

[29] Fowler, M. *Domain-specific languages*. Addison Wesley, 2010. ISBN: 978-0321712943.

[30] Garfinkel, S. Digital forensics research: The next 10 years. In *Proceedings of the 2010 DFRWS Conference* (2010). 10.1016/j.diin.2010.05.009.

[31] Garfinkel, S. Digital forensics xml and the dfxml toolset. *Digital Investigation 8*, 3 (2012), 161 – 174.

[32] Garfinkel, S., Farrell, P., Roussev, V., and Dinolt, G. Bringing science to digital forensics with standardized forensic corpora. In *Proceedings of the Ninth Annual Digital Forensic Research Conference* (2009), DFRWS, pp. S2–S11. 10.1016/j.diin.2009.06.016.

[33] Garfinkel, S., Malan, D., Dubec, K.-A., Stevens, C., and Pham, C. *Advanced Forensic Format: an Open Extensible Format for Disk Imaging*. Springer New York, Boston, MA, 2006, pp. 13–27.

[34] Garfinkel, S., Nelson, A. J., and Young, J. A general strategy for differential forensic analysis. In *12th Annual Digital Forensics Research Conference* (2012), DFRWS'12, pp. S50–S59. 10.1016/j.diin.2012.05.003.

[35] Garfinkel, S. L. Digital media triage with bulk data analysis and bulk_extractor. *Journal of Computers & Security 32* (2013), 56–72. 10.1016/j.cose.2012.09.011.

[36] Gerber, M., and Leeson, J. Formalization of computer input and output: the hadley model. *Digital Investigation 1*, 3 (2004), 214 – 224.

[37] Ghosh, D. *DSLs in Action*, 1st ed. Manning Publications Co., 2010. ISBN: 978-1935182450.

[38] Gladyshev, P., and Patel, A. Finite state machine approach to digital event reconstruction. *Digital Investigation 1*, 2 (2004), 130–149. 10.1016/j.diin.2004.03.001.

[39] Grajeda, C., Breitinger, F., and Baggili, I. Availability of datasets for digital forensics – and what is missing. *Digital Investigation 22* (2017), S94 – S105.

[40] Group, T. C. D. E. S. F. W. Standardizing digital evidence storage. *Commun. ACM 49*, 2 (Feb. 2006), 67–68.

[41] Guo, Y., Slay, J., and Beckett, J. Validation and verification of computer forensic software tools—searching function. *Digital Investigation 6* (2009), S12 – S22. The Proceedings of the Ninth Annual DFRWS Conference.

[42] Harvey, P. ExifTool by Phil Harvey. http://www.sno.phy.queensu.ca/~phil/exiftool/.

[43] Hibshi, H., Vidas, T., and Cranor, L. Usability of forensics tools: A user study. In *2011 Sixth International Conference on IT Security Incident Management and IT Forensics* (May 2011), pp. 81–91.

[44] Hitchcock, B., Le-Khac, N.-A., and Scanlon, M. Tiered forensic methodology model for digital field triage by non-digital evidence specialists. *Digital Investigation 16* (2016), S75 – S85. DFRWS 2016 Europe.

[45] Kahvedžić, D., and Kechadi, T. Dialog: A framework for modeling, analysis and reuse of digital forensic knowledge. *Digital Investigation 6* (2009), S23 – S33. The Proceedings of the Ninth Annual DFRWS Conference.

[46] Kamp, P.-H., and Watson, R. N. M. Jails: Confining the omnipotent root. In *Proceedings of the Second International System Administration and Networking Conference* (2000), SANE. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.3596&rep=rep1&type=pdf.

[47] Karen Kent, Suzanne Chevalier, T. G. H. D., 2006. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-86.pdf.

[48] Kryder's law, 2015. https://www.scientificamerican.com/article/kryders-law/.

[49] Levine, B. N., and Liberatore, M. Dex: Digital evidence provenance supporting reproducibility and comparison. *Digital Investigation 6* (2009), S48 – S56. The Proceedings of the Ninth Annual DFRWS Conference.

[50] Lewis, N., Case, A., Ali-Gombe, A., and Richard, G. G. Memory forensics and the windows subsystem for linux. *Digital Investigation 26* (2018), S3 – S11.

[51] Ligh, M. H., Case, A., Levy, J., and Walters, A. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1 ed. Wiley, 2014. ISBN: 978-1118825099.

[52] Linux 3.8, 2013. https://kernelnewbies.org/Linux_3.8.

[53] LXC – Linux Containers, 2008-17. https://github.com/lxc/lxc.

[54] Love, C. Digital: A love story, 2010.

[55] LXD – Linux Containers. https://linuxcontainers.org/lxd.

[56] Menage, P. B. Adding generic process containers to the Linux kernel. In *Proceedings of the Ottawa Linux Symposium* (2007), pp. 45–58. https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf.

[57] Merkel, D. Docker: Lightweight Linux Containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014).

[58] Mernik, M., Heering, J., and Sloane, A. M. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys 37*, 4 (2005), 316–344. 10.1145/1118890.1118892.

[59] Merriam-Webster. Cybercrime.

[60] MITRE. Cce - common event expression. http://cee.mitre.org/, 2014. Accessed: 2019-03-25.

[61] Moser, A., and Cohen, M. I. Hunting in the enterprise: Forensic triage and incident response. *Digital Investigation 10*, 2 (2013), 89 – 98. Triage in Digital Forensics.

[62] NIST, 2017. https://www.nist.gov/itl/ssd/software-quality-group/computer-forensics-tool-testing-program-cftt.

[63] NIST, 2019. https://toolcatalog.nist.gov.

[64] Odersky, M., Spoon, L., and Venners, B. *Programming in Scala: Updated for Scala 2.12*, 3rd ed. Artima Press, 2016. ISBN: 978-0981531687.

[65] of Justice Technical Working Group, U. S. N. I. A guide for first responders. *Electronic Crime Scene Investigation* (2001).

[66] OSI Model. http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip.

[67] Parr, T. ANTLR. http://www.antlr.org/index.html.

[68] Parr, T. *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013. ISBN: 978-1934356999.

[69] Pike, R. Generics - problem overview, Aug 2018.

[70] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13* (2015), 277–298.

[71] Pirolli, P., and Card, S. 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* (2005). http://researchgate.net/publication/215439203.

[72] Puppet Labs. Pup-987 remove Ruby DSL support, 2015. https://tickets.puppetlabs.com/browse/PUP-987.

[73] Reith Mark, Carr Clint, G. G. An examination of digital forensic models. *International Journal of Digital Evidence* (2002).

[74] Richard, G., and Roussev, V. Next-generation digital forensics. *Communications of the ACM 49*, 2 (Feb 2006), 76–80. 10.1145/1113034.1113074.

[75] RKT – Linux Containers. https://coreos.com/rkt.

[76] Roussev, V. Building a forensic computing language. In 48*th* *Hawaii International Conference on System Sciences* (2015). 10.1109/HICSS.2015.617.

[77] Roussev, V. *Digital Forensic Science: Issues, Methods, and Challenges*, 1 ed. Morgan & Claypool Publishers, 2016. ISBN: 978-1627059596.

[78] Roussev, V., Ahmed, I., Barreto, A., McCulley, S., and Shanmughan, V. Cloud forensics – tool development studies and future outlook. *Digital Investigation 18*, Suppl C (2016), 79–95. 10.1016/j.diin.2016.05.001.

[79] Roussev, V., Quates, C., and Martell, R. Real-time digital forensics and triage. *Digital Investigation 10*, 2 (2013), 158–167. 10.1016/j.diin.2013.02.001.

[80] Roussev, V., and Richard, G. Breaking the performance wall: The case for distributed digital forensics. In *Proceedings of the 2004 Digital Forensic Research Workshop* (2004), DFRWS. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.8692&rep=rep1&type=pdf.

[81] Roussev, V., Wang, L., Richard, G., and Marziale, L. *A Cloud Computing Platform for Large-Scale Forensic Computing*. Springer Berlin Heidelberg, 2009, ch. 6, p. 15. http://dx.doi.org/10.1007/978-3-642-04155-6_15.

[82] Sammons, J. *The basics of digital forensics: the primer for getting started in digital forensics*. Elsevier, 2015.

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

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

[85] SHAW, A., AND BROWNE, A. A practical and robust approach to coping with large volumes of data submitted for digital forensic examination. *Digital Investigation 10*, 2 (2013), 116 – 128. Triage in Digital Forensics.

[86] SOFTLEY, I. Hackers, 1995.

[87] SOMMER, P. Forensic science standards in fast-changing environments. *Science & Justice 50*, 1 (2010), 12 – 17. Special Issue: 5th Triennial Conference of the European Academy of Forensic Science.

[88] STELLY, C., AND ROUSSEV, V. Scarf: A container-based approach to cloud-scale digital forensic processing. *Digital Investigation 22* (2017), S39 – S47.

[89] STELLY, C., AND ROUSSEV, V. Nugget: A digital forensics language. *Digital Investigation 24* (2018), S38 – S47.

[90] STELLY, C., AND ROUSSEV, V. Language-based integration of digital forensics & incident response. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (New York, NY, USA, 2019), ARES '19, Association for Computing Machinery.

[91] STEWART, J. Scalable forensics with TSK and Hadoop. In *Open Source Digital Forensics Conference* (2012), OSDFCon. https://www.osdfcon.org/presentations/2012/OSDF-2012-The-Sleuth-Kit-and-Apache-Hadoop-Jon-Stewart.pdf.

[92] STOLL, C. *The Cuckoos Egg: tracking a spy through the maze of computer espionage*. Pocket Books, 2005.

[93] STOLL, C. *CUCKOO'S EGG*. Doubleday, 2012.

[94] STÜTTGEN, J., AND COHEN, M. Robust linux memory acquisition with minimal target impact. *Digital Investigation 11* (2014), S112 – S119. Proceedings of the First Annual DFRWS Europe.

[95] THE APACHE TEAM. *Apache Hadoop MapReduce*. https://hadoop.apache.org/.

[96] THE APACHE TEAM. *Apache Pig*. https://pig.apache.org/.

[97] THE GOLANG TEAM. Go. https://golang.org/.

[98] TIOBE. Tiobe index for october 2018. https://www.tiobe.com/tiobe-index/.

[99] U.S. DEPARTMENT OF JUSTICE, OFFICE OF THE INSPECTOR GENERAL. Audit of the Federal Bureau of Investigation's Philadelphia Regional Computer Forensic Laboratory, 2015. https://oig.justice.gov/reports/2015/a1514.pdf.

[100] U.S. DEPARTMENT OF JUSTICE, OFFICE OF THE INSPECTOR GENERAL. Audit of the Federal Bureau of Investigation's New Jersey Regional Computer Forensic Laboratory, 2016. https://oig.justice.gov/reports/2016/a1611.pdf.

[101] VAN BAAR, R., VAN BEEK, H., AND VAN EIJK, E. Digital forensics as a service: A game changer. *Digital Investigation 11* (2014), S54 – S62. Proceedings of the First Annual DFRWS Europe.

[102] van Beek, H., van Eijk, E., van Baar, R., Ugen, M., Bodde, J., and Siemelink, A. Digital forensics as a service: Game on. *Digital Investigation 15* (2015), 20 – 38. Special Issue: Big Data and Intelligent Data Analysis.

[103] Wirth, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of ACM 20*, 11 (Nov 1977), 822–823. 10.1145/359863.359883.

Vita

Christopher Drew Stelly was born in Mobile, Alabama. He completed his undergraduate degree of computer science at the University of Louisiana in 2012 as a distinguished senior. His notable projects included an autonomous vehicle for a DARPA challenge, an AI-based ant colony, and a picosatellite which was launched into orbit in 2013. After re-enrolling into the graduate program, he transferred from UL to the University of New Orleans, where he completed his master's degree of computer science in 2013. His thesis, entitled "Dynamic Aspect Oriented Bytecode Instrumentation", showcased how bytecode based mobile applications could be injected with specific code for both beneficial and malicious intent. After a semester break, he enrolled in UNO's PhD of Engineering and Applied Science program, where he is researching digital forensics and cyber security under Dr. Vassil Roussev. Their research has been published multiple times and has earned an NSF grant.

In 2012, Stelly began working for Lockheed Martin. Since 2014, he has worked for the corporation's cyber testing and exploitation team. The team is responsible for performing adversarial themed cyber testing, or "red teaming", against all programs and platforms under the Lockheed Martin umbrella.