University of Nevada, Reno

AKF: A modern synthesis framework for building digital forensic datasets

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Computer Science and Engineering

by

Lloyd Gonzales

Nancy LaTourrette, Advisor
May 2025

Abstract

As our world becomes increasingly dependent on technology, the advancement of digital forensics has become a key focus in the fight against cybercrime. The field depends greatly on the availability of disk images, network captures, and other forensic datasets for education, tool validation, and research. However, real-world datasets often contain sensitive information that may be difficult to remove, making them difficult to distribute publicly. As a result, researchers and educators can encounter gaps in available datasets, often leading to the manual development of new, suitable datasets. While viable, this approach is time-consuming and rarely produces datasets that accurately reflect real-world scenarios suitable for comprehensive training and education. In turn, there is ongoing research into forensic synthesizers, which automate the process of creating unique synthetic datasets that can be publicly distributed without legal and logistical concerns.

This thesis introduces the *automated kinetic framework*, or AKF, a modular synthesizer for creating and interacting with virtualized environments to simulate user activity. AKF significantly improves upon the architectural designs of prior synthesizers while maintaining feature parity and usability. Additionally, AKF leverages the CASE standard to provide human- and machine-readable reporting, exposing low-level details in a searchable format. Finally, AKF provides options for leveraging generative AI to develop high-level scenarios as well as individual artifacts. These contributions are intended to not only improve the speed at which synthetic datasets can be created, but also ensure the long-term usefulness of AKF-generated datasets and the framework as a whole.

Dedication

To those in the osu! tournament community, without whom I would have never embarked on this journey;

To my numerous teachers and professors, especially Keith Lightfoot, Rodney Rogers, Marc Miller, and Gabbi Bachand, who I have limitless and appreciation and admiration for;

To those part of the United States Cyber Team and the broader CTF community, for igniting my interest in digital forensics and supporting me even when I flailed like a fish out of water;

And, of course, to my friends, family, and bed, who provided motivation when there was none.

Acknowledgments

I want to express my immense gratitude to Nancy Latourrette for her support, guidance, and mentorship throughout the development of this thesis. This thesis would be nowhere without her ideas and experience, and I am truly grateful and honored to have been able to work with her throughout this experience.

I would also like to thank Bill Doherty for his review of a prior paper, from which some of this content is derived.

Table of Contents

List of Tables

List of Figures

List of Code Listings

Chapter 1

Introduction

1.1 History of digital forensics

Digital forensics is a relatively new field originating from the growing need to address computer crimes. Over the last 30 years, computers have become critical in virtually every modern industry. In turn, they have become the target – and weapon – of many attacks. Much like conventional crimes, cybercrimes leave behind traces of digital evidence that can be analyzed to determine the nature of the incident, the scope and extent of the crime, and more. Today, digital forensics is also applicable to a variety of other contexts, such as incident response and corporate investigations, further highlighting the need for quality educational material and hands-on training for new forensic analysts.

The origins of digital forensics can be traced to the 1980s, during which computers began to be adopted by the general public. Prior to this, computers were restricted mainly to industry, academia, and governments with dedicated infrastructure and staff [1]. However, with the introduction of PCs that were more accessible to the

typical consumer, such as the Commodore 64 and the IBM PC, computer usage within the public grew. So did crimes committed with computers; people discovered that they could hack telephone networks to illegally obtain software and "free" telephone services [2]. In 1983, Canada would be the first government to amend its criminal code to cover cybercrime; the United States, the United Kingdom, and Australia would follow suit in the following decade.

During this time, investigations were relatively ad-hoc and very simple. Teams often built their own software to analyze devices, though a few hobbyists and software vendors began to develop dedicated forensic tools. However, computers had yet to truly enter the mainstream; most individual computer owners were (relatively wealthy) computer hobbyists rather than the general public. In 1984, the U.S. Census Bureau determined that only about 8.2% of U.S. households owned a computer [3], nearly doubling to 15% in 1989 [4].

As technology continued to advance past the 1990s, digital forensics – and computing as a whole – began to grow in scope and importance. The rise of mobile devices and the internet drastically changed the role of computing in the eyes of the public; with it came the rise of cybercrime and a recognition of the importance of computer investigations. These early investigations continued to be performed by investigators who happened to be experienced with computers rather than those with formal digital forensics training [5]. Such investigations lacked a standardized structure (unlike the formal approaches in "traditional" forensic disciplines), contributing to debates about the reliability of digital forensics [6].

By the turn of the century, digital forensics had grown beyond law enforcement and niche cases to becoming a focus of broader research and education. The first use of the phrase "computer forensics" in ACM literature appeared in 1999 [7], with the first Digital Forensic Research Workshop (DFRWS) held in 2001 to identify priorities

in the growing field of digital forensics [8]. A particularly notable case was that of the September 11, 2001 attacks, in which computers were found to contain significant evidence related to the organization and planning of the attack. Intelligence communities and law enforcement agencies worldwide began establishing digital forensics teams, demonstrating a shift in which investigations previously only conducted by individuals and small teams were now also performed by governments and professional organizations.

It was during this time that tools such as EnCase and Forensic Toolkit (better known as FTK) evolved to become dedicated products that remain a mainstay of the digital forensics field today [1]. At the same time, anti-forensics began to grow in popularity; numerous tools and resources were developed with the express goal of exploiting and hindering the digital forensics process, generally with the stated motivation of guarding users' privacy and protecting users from punishment for undesirable computer activity [9], [10].

By this time, it had become abundantly clear that there needed to be dedicated training to develop specialists in digital forensics. Undergraduate and graduate curricula dedicated to the study of digital forensics were developed [11], [12] along with numerous efforts to standardize and improve digital forensics curricula [7], [13], [14], [15], [16]. The development of formal education was significant due to the growing importance of digital forensics from a legal perspective; analysts were expected to follow a strict procedure to ensure the admissibility of digital evidence into a court of law [17]. Simultaneously, analysts needed sufficient experience to provide an unbiased, accurate opinion of this digital evidence in court as an expert witness.

Digital forensics continues to be an important focus in industry. The Bureau of Labor Statistics projects that information security employment will grow 32% over the next decade from 2022 to 2032, adding over 50,000 new jobs [18]. The diversity

and depth of digital forensics will continue to grow with sustained developments in modern computing – a fact that extends beyond industry and into research and education.

With the growing importance and complexity of digital forensics, there is a clear need for high-quality, realistic data for researchers and instructors alike. However, there continue to be significant gaps in the quantity and variety of material suitable for digital forensic training; in particular, privacy and legal concerns prevent the use and distribution of real-world data [19]. From a research perspective, the result is that many researchers develop their own datasets, often with a very narrow scope and a low degree of reproducibility [20], [21]. From an education perspective, the result is that most training material is either manually created by instructors or reused from existing sources. That is, researchers and instructors alike often create their own datasets because publicly available corpora are insufficient; however, this is a time-consuming process that responds slowly to changes in technology and software.

There have been various efforts to automate and streamline the process of creating new forensic datasets from high-level descriptions and predefined forensic artifacts. These forensic synthesis frameworks, also known as "synthesizers," include various features geared toward rapidly developing datasets for research and education. These features include mass dataset generation for large classrooms, metadata generation useful in tool validation, and the ability to generate and export a variety of forensic artifacts from the synthesizer. This is particularly valuable in classrooms, as these datasets enable instructors to create images that allow students to learn about a specific forensic technique while emulating some real-world challenges that forensic analysts face in practice.

However, there is still much work to be done towards increasing the accessibility and flexibility of these frameworks. Before exploring synthesizers in greater detail, it

is necessary to first understand the purpose and characteristics of forensic datasets in a variety of contexts. Doing so will not only outline *why* the development of synthesizers is necessary but also *what* features these synthesizers must provide. Once we have established this foundation, we can begin exploring *how* a synthesizer should be architected – the focus of the remainder of this thesis.

1.2 Purpose of forensic datasets

1.2.1 In industry

Before considering the use of forensic datasets in research and education, we begin by exploring how these datasets are acquired and used in the "real world" – that is, in the context of investigations conducted by professionals in industry.

In practice, forensic datasets – and digital forensics as a whole – are used for a variety of purposes. In particular, Conklin et al. identify three primary cases in which digital forensics may be performed [17]:

- **To investigate computer systems related to a violation of law**: This includes cases regarding the distribution and storage of illegal content, the use of a computer to launch denial of service attacks against an individual or organization, and the proliferation of harmful malware within an organization.
- **To investigate computer systems for compliance (or a violation of) an organization's policies**: This primarily covers internal investigations in which a user may not have broken local legislation but may have violated a company policy. For example, many companies restrict access to computing systems based on the time of day as a security measure. Although a user may typically have authorization to access the organization's network, unexpected

weekend activity may require forensic analysis to determine if any actions were done with malicious intent.

- **Responding to a legal (or internal) request for digital evidence**: This process is known as e-discovery, in which an organization preserves and produces digital information typically as part of the discovery process in lawsuits. With court approval, organizations can be compelled to turn over relevant information to a particular lawsuit, including digital documents and digital artifacts such as file metadata. (The Federal Rules of Civil Procedure were amended in December 2006 to include "electronically stored information" as part of civil discovery [22].)

Additionally, it is important to note that the acquisition of forensic evidence is only one part of the overall digital forensics process. For example, analysts must consider the order in which to obtain evidence while adhering to chain of custody procedures. Conklin et al. identify several steps throughout the lifespan of a digital forensic investigation that are summarized here [17]:

- **Identification**: While rarely covered by educational datasets, it is important to determine the scope of the devices that must be analyzed for an incident. It is impossible to investigate an incident until an organization can ascertain that one has occurred (which requires implementing detection and protection practices, such as those described in the NIST Cybersecurity Framework [23]). Similarly, if the scope of an incident is poorly defined, the subsequent forensic investigation may fail to find relevant evidence. Scoping not only impacts the investigation's success but also the organization's response as a whole.
- **Preservation**: After identifying the relevant machines, analysts must secure and preserve the physical device itself. With guidance from the organization and an analyst's judgment, this often involves prioritizing the devices that must

be imaged first; for example, a critical server may be more likely to cycle out important logs first, or an employee's device may only hold important information in volatile memory. Such devices should be prioritized over those that are unlikely to lose relevant information if not imaged immediately.

- **Collection**: At this point, an analyst must now duplicate the digital evidence and any relevant physical evidence. Collection must be performed in a manner that passes legal scrutiny; that is to say, it must meet requirements for accuracy, reliability, and relevance [17], [20] (in the United States). In the case of disk imaging, this is typically done with a write blocker and cryptographic hashing algorithms to ensure a faithful copy of nonvolatile memory has been created.
- **Analysis**: Here, an analyst uses tools and their own knowledge to identify significant pieces of information within collected images, reconstructing data fragments and drawing conclusions to form a coherent timeline and scenario. In many cases, this is done using tools such as Sleuth Kit, Autopsy, EnCase, and other domain-specific software [2], which often parse and automatically identify data of interest on a reconstructed file system.
- **Reporting**: After an analyst (or a team of analysts) has completed their analysis of the collected data, they must summarize and provide a non-technical overview of the conclusions drawn from the investigation.

Throughout this process, analysts must adhere to well-established norms to ensure the admissibility of any digital evidence in court. For example, chain of custody must be maintained, detailing access and movement of any evidence. This applies to "conventional" and digital forensics alike, as this documentation asserts that the evidence has not been tampered with as it is handled between analysts and locations. Without proper documentation, such as a chain of custody, critical evidence may not meet the legal requirements for admissibility, changing the outcome of a trial.

A forensic investigation is often only part of a larger incident response effort. For example, if a data breach occurs, a digital forensic investigation may be used to determine the scope and methodology of the breach itself. However, other teams within the organization may be responsible for patching the devices responsible for the breach, determining legal consequences, and engaging in other recovery-related activities. Additionally, a forensic analyst may have non-analytic responsibilities; for example, in addition to developing a written report, they may be called as an expert witness to testify and defend conclusions drawn from the investigation [7], [11], [17].

In the context of research and education, most hands-on or practical training typically focuses on the analysis and reporting steps rather than the complete investigation process. That is to say, although the theory provided in training covers this entire process in detail, it is relatively rare for students to secure data from an organization, use a write blocker and other hardware needed to image devices, and then present the conclusions as an expert witness [7]. While experience here is important, it is often impractical; not every university will have a large forensics lab or a courtroom regularly available. Additionally, the techniques of analysis and reporting are arguably the most important; while the other steps can be learned "on the job" relatively quickly, all students must be familiar with modern software and tooling, as well as recent advancements in forensic and anti-forensic techniques. As a more concrete example, deep technical knowledge of operating systems is not required to acquire disk images but is undoubtedly required for effective analysis and reporting.

It is for this reason that the vast majority of education and research focuses primarily on the analysis step. Improvements to the distribution and creation of training material – such as the use of online labs and synthesizers – are primarily driven by the fact that physical labs are not necessary, allowing institutions to instead focus on the software and skill set needed to analyze images and draw conclusions

effectively [24], [25].

1.2.2 In research

In many cases, the focus of research in digital forensics is on improving specific processes in the analytic step of a forensic investigation. This includes the development of analysis techniques for niche platforms, direct improvements to existing techniques, or novel methodologies for performing forensic investigations for a particular platform. For example, recent publications from the Digital Forensic Research Workshop detail a new hashing technique for detecting similarities in arbitrary files [26], an analysis of the NAND memory of the Nintendo 3DS [27], and the development of a new tool for the automated analysis of Android mobile devices [28]. Necessarily, the research in each of these publications requires forensic datasets; in most cases, these are obtained or developed manually by the authors, as opposed to using an existing public or private dataset.

Besides novel contributions to the field, other research focuses on upholding the quality of the investigation process as new technologies and tools to analyze datasets are developed. For example, NIST maintains the *Computer Forensics Tool Testing* (CFTT) program, which provides a standard methodology and test corpora for evaluating specific forensic tool capabilities [29]. Specific capabilities tested include the ability of a tool to perform string searching, disk imaging, and the recovery of deleted files. The project also includes catalogs for forensic algorithms, software, and tools, as well as the *Computer Forensic Reference Data Sets* (CFReDS) project, which is a repository of forensic datasets contributed by NIST and other organizations that is often used by both instructors and researchers [30].

1.2.3 In education

Finally, we address the generation of forensic datasets as it applies to education. Datasets in an educational setting typically cover a range of techniques and tools, allowing students to practice applying theoretical concepts learned through lectures and recitations [31]. Often, this is done in the context of a specific scenario, such as a user stealing files from a protected company server, using steganography to hide information, or recovering a file from volatile memory. Besides the specific technical skills covered by these scenarios, these datasets aim to develop the analytical skills needed for students to adapt to developments in tools and technology [7]. In other words, students should be familiar with common tools and patterns in digital forensics, providing a foundation on which more niche techniques can be learned [25].

Although the focus of individual forensic datasets is often to improve the skills of students in the analysis phase, these images have a direct impact on the reporting phase as well. Indeed, students must be able to accurately summarize their conclusions, using their judgment to describe a scenario and identify topics of interest in a manner consistent with the law. Although important, digital forensics is not just the effective use of tools and techniques; a student should also aim to be well-rounded in the legal, social, and professional aspects of digital forensics [11].

The challenge, however, is providing hands-on labs that comprehensively support the ideas learned in theoretical courses. Indeed, instructors face numerous challenges when providing realistic lab material, including limited access to the necessary software and hardware; the extensive time needed to develop, distribute, and grade labs; and the high variability between different forensics programs [25], [31], [32]. Much of this difficulty in providing high-quality images can be traced to the issues associated with acquiring both “real” and “synthetic” datasets for educational purposes, as described in the following section.

1.3 Real and synthetic datasets

Now that we have identified the purpose of forensic datasets and why they are needed, we now move to a discussion of how these datasets are acquired, as well as various issues encountered when using these datasets. This section focuses on the qualities of real and synthetic datasets, including some examples. A more detailed survey of existing datasets is presented in section 2.1.

There are two types of forensic datasets described by Park [33] based on the original taxonomy described by Garfinkel et al. [20]. The first is "real" data, which is organically created by humans without the explicit intent of being used in a forensic investigation. The other is "synthetic" data, which is generated for specific educational and research purposes. Synthetic datasets are considerably more common than real datasets in education for the reasons discussed in the following sections, even if they require significantly more effort to develop.

It should also be noted that this section provides a general overview of forensic datasets as they relate to education and research, providing the necessary context for forensic synthesizers. However, Horsman and Lyle have performed a more exhaustive description of dataset construction and usage [34].

1.3.1 Real datasets

Real data is inherently the most "realistic" form of forensic data, containing extensive background noise as a result of typical long-term device usage in addition to a variety of software, operating system artifacts, and other files that might be of interest in a real forensic investigation. Real datasets are most reflective of the scenarios that industry professionals face, allowing students to train themselves in separating relevant content from irrelevant content while identifying and synthesizing details from both

a system and a human perspective.

Real data can be sourced from far more than just the hard drives of computers. Other potential and previously used data sources include:

- social media (which often contains a variety of artifacts with revealing metadata and open source information) [35];
- packet sniffers and dedicated forensic tools for IoT devices [36];
- video game consoles [21], [27];
- cloud web server logs [37];
- honeypots [38];
- and the Apache and Python mailing archives [21].

Many more public forensic repositories are described by Grajeda et al., though just as many datasets used in research remain private [21]. With the billions of internet-connected devices today, there should be no shortage of sources for real datasets. However, real datasets are the most challenging to work with in education for a variety of reasons, particularly the legal and privacy concerns surrounding the use and distribution of these datasets, as well as their broad scope and lack of prior analysis.

From a scoping perspective, real datasets were not created to educate students about a particular technique and may not adequately supplement an instructor's material without significant effort. For example, Garfinkel identifies several requirements for forensic datasets to be suitable for various uses in research and industry [19]. Garfinkel notes that in addition to the overall lack of publicly available corpora, many real datasets suffer from a lack of complexity, supplemental annotations, or ongoing maintenance – all of which are often needed in an educational context and cannot easily be added to real datasets after the fact.

With respect to availability, some of these datasets are inherently publicly available; in other cases, their access is restricted to within an organization. However, just because the underlying data is public does not necessarily mean that it exists in an aggregate form immediately suitable for research while preserving individuals' privacy. Even when aggregated, concerns arise from the use of public datasets for unintended purposes, such as the use of social media as a source of forensic data; these questions are already the focus of debate in generative AI, which uses publicly available data to train models for both commercial and research use [39], [40], [41].

That said, the primary barriers to the use of real forensic datasets are privacy and legal issues. For example, between 1998 and 2006, Garfinkel acquired over 1,000 hard drives through secondary markets, allowing researchers to perform a range of studies on a large set of real-world data [19]. Although legal for use by private institutions, the same dataset was barred from use at the Naval Postgraduate School due to concerns with federal privacy legislation. Similarly, Grajeda et al. noted that nearly half of all digital forensic literature reviewed using a novel dataset did not publish the dataset due to legal restrictions or NDAs. This was often because the datasets were obtained from government agencies, corporations, and law enforcement agencies and could not be publicly released [21]. Another reason is that the dataset may contain objectionable or illicit material, such as licensed software or pornography, preventing public distribution.

Some organizations and researchers have made efforts to make real datasets accessible to the public through various means. For example, the emails seized during the Federal Energy Regulatory Commission's investigation of Enron were purchased by the Massachusetts Institute of Technology, which anonymized emails and attachments before distributing the dataset to the public [19], [42]. In other cases, institutions have aimed to ensure the data can be made publicly available ahead of time, such as by

requesting that individuals sign an agreement before any data is collected.

Indeed, some educational institutions use real-world datasets in forensic labs. Besides the sources mentioned above, students may also opt to image their own devices or the device of a friend with permission; these approaches are mentioned in older works [11], [38], prior to the advent of online platforms such as CFReDS that provided easier access to education-focused datasets. Naturally, this method of acquiring real-world images suffers from other issues as well; students know exactly what they will find on their own computer, and although individuals may consent to the use of their images for educational purposes, there remains the risk of highly personal data being leaked as a result of the exercise [20]. Simultaneously, they often lack the ground truth or annotations needed to assert that a student has found everything of interest.

1.3.2 Overview of synthetic datasets

Because of the various issues associated with real-world data, many instructors use synthetic data (or "manually created data") instead, in which the scenario and data are artificially generated based on some predetermined procedure. That is to say, the data exists with the explicit intent of being used in research or education. More precisely, synthetic datasets are created to accurately reflect the challenges faced by industry professionals while avoiding the privacy and legal restrictions of using genuine, real-world scenarios.

As described by Park and Garfinkel et al., synthetic data can be categorized into two distinct groups [20], [33]:

- **Synthetic test data**, which refers to forensic corpora that have been developed to test specific features in a group of tools. These are well-annotated datasets with extensive reference information and ground truth data, typically used to

assert that a tool is able to analyze and identify data of interest. One example discussed so far is the Computer Forensics Tool Testing program administered by NIST [29].

- **Synthetic realistic data**, which is designed to mimic a situation that a forensic examiner might encounter in a real-world investigation. These are typically more applicable to an educational context than test data, though test data can be used as educational material for learning new tools or particular techniques.

Synthetic realistic data varies greatly in scope. Simple datasets might form a realistic emulation of how a real-world user might use a particular piece of software or anti-forensic technique, which creates forensic artifacts directly associated with these actions. More complex datasets can form a complete simulation of a scenario, whether based on actual events (such as the one developed by Moch and Freiling based on the Arno Funke blackmail case in Germany [38]) or on common industry themes, such as the NIST CFReDS data leakage case [43].

1.3.3 Motivation for synthetic datasets

Besides the issues mentioned in subsection 1.3.1, one primary motivation for the development of synthetic datasets is the need to fill various gaps in publicly available corpora. The survey done by Grajeda et al. found that some researchers created new datasets simply because there was no available dataset for their needs – not because of legal or privacy concerns [21]. The researchers interviewed in the survey also recognized the value of publishing their datasets but were often hindered by a lack of available resources to maintain or distribute the datasets. Other researchers were prevented from doing so due to a non-disclosure agreement. The survey concluded that there was no preference for building new datasets from scratch in research, though there was clear agreement that private datasets made it more difficult for researchers

to reproduce results; in general, publishing datasets contributed to the advancement of the field.

Similar sentiment also exists in an educational context, though the motivation for creating new images from scratch differs from that of research. For example, the direct reuse of existing datasets is sometimes undesirable, as it is often the case that answer keys and walkthroughs have already been published online [44]; additionally, students in the same class can simply replicate the exact methodology of other classmates to come to the same conclusions without needing to perform meaningful analysis. Even in labs where students' actions can be monitored to provide insight into their methodology and possible mistakes, such as in the *CYDEST* platform developed by ATC-NY [24], it is still possible to copy the methodology of another student if the labs themselves do not differ.

Additionally, some datasets that reflect real-world scenarios are unsuitable for academic purposes. For example, the forensic challenges of the Honeynet Project, DFRWS, and DC3 are suitable training material for experienced forensic analysts but are often too difficult for students to solve [44]. As mentioned before, many real-world datasets lack "ground truth" or annotated information that is suitable for determining the contents of a dataset, preventing the development of an answer key and qualitative grading of students' work based on what they find and report. The same is true of these synthetic datasets, which may have been developed without a focus on detailed documentation.

Clearly, there is a need for datasets that allow students to explore the techniques and tools used by forensic analysts in industry in a realistic setting. These datasets should be built such that they challenge students to explore the same techniques but have distinguishable differences to dissuade cheating. For example, data may be written to a different disk sector, relevant metadata may be changed, and files not

relevant to a scenario may be modified. Simultaneously, instructors need to know what is contained in these datasets (including the presence of variations) to judge the accuracy of students' findings and determine which images are suitable hands-on material to complement lecture material.

As a result, many instructors ultimately turn to manually developing realistic forensic scenarios that students can analyze, much like researchers. Some of these efforts have led to publicly available datasets that encompass realistic scenarios, such as the datasets and scenarios published by Woods et al. as a direct response to the issues noted above [44].

1.3.4 Challenges in developing synthetic datasets

However, fulfilling all of the ideal characteristics of a forensic dataset is difficult, especially if an instructor wants to create multiple variations of the same forensic scenario, further multiplied by the number of scenarios presented throughout a course. There are three distinct issues associated with the manual development of forensic datasets: the significant amount of time involved in their creation, the inherent non-determinism of human creation, and the lack of realistic background usage.

The primary issue associated with manual development is the extensive amount of work and time that must be put into not only planning and developing the scenario but also executing it. The time-consuming process involved in the manual development of disk images suitable for education is widely recognized [31], [32], [38], [44], [45], [46], [47], motivating research into possible solutions to streamlining their development.

One such solution is to "falsify" metadata as a means of speeding development up. In particular, it is possible to directly change the time in virtualization software (the approach often used by synthesizers [38], [45]) to allow time to pass without leaving

any artifacts that suggest that the system time has been tampered with. However, it is often challenging to produce the artifacts associated with a scenario over a shorter period of time than is suggested by the metadata attached to the artifacts. For example, modifying metadata to be consistent with online services can be challenging since artifacts such as emails or cached external websites might be based on "real" time; additional work might be needed to synchronize online content typically outside of the control of the instructor. While there are workarounds that could be considered – such as saving the contents of various external websites and then "replaying" them at the time of scenario development – these are not trivial.

The net result is that a scenario implied to take place over several days will also require several days of effort to create if done manually. This extended development period also comes with the challenge of addressing mistakes. If an unintended action is performed while constructing a dataset, instructors have several options. Documenting the deviation may be sufficient; in other cases, these must be corrected by directly editing the virtual hard drive. This was the approach taken by Woods et al. in which at least one researcher logged into their personal email account while developing a fictional scenario; the resulting disk images were later scanned for personal identifiers and stripped as needed [44]. However, consider a scenario in which it is essential that specific files are deleted and written in a particular order, perhaps to ensure the operating system behaves in an expected manner. In this case – and possibly many others – the only option may be to restart the process of developing the dataset.

Furthermore, the non-determinism associated with human creation is a double-edged sword. Given the same general scenario, this allows multiple images to differ very slightly in nature (a desirable feature as described earlier in this section). For example, in the scenario developed by Woods et al., the scenario developers performed

actions according to a predefined timeline. However, these developers did not carry out these actions at the same time or in the same manner [44]. This provides the variability needed to prevent students from directly copying answers while allowing students to explore the same techniques at a high level.

However, this also adds uncertainty to the images. The existence of certain artifacts within the datasets may vary due to confounding factors in the procedure used to create them. For example, suppose that two researchers follow the same procedure to delete and overwrite several files, creating a disk image suitable for testing file carving techniques from slack space. Due to variations in operating systems, drivers, and other related software, the recoverable contents in slack space may differ significantly between the two researchers. Similarly, consider a case in which two researchers follow the same procedure to develop volatile memory captures on a Linux machine. It might be the case that the paging daemon evicts relevant data for one image but not the other, causing significant differences. Without careful preparation to create a consistent environment for development and testing, these differences might arise without a clear reason. Where possible, sources of uncertainty should be minimized or at least known to the scenario developer.

Finally, an inherent limitation of the long development time needed to produce these images is the difficulty in creating realistic background noise, in which a user performs normal activities on the computer that are entirely unrelated to the meaningful elements of the scenario. Background noise must be generated in a way that is realistic to the scenario while avoiding any activity that could personally identify the researcher or instructor. Although some background noise can be automated – such as browsing predefined websites or sending predefined emails – the fact remains that many realistic "background" activities on GUI-based operating systems are not easy to perform without a human. Furthermore, background noise may comprise a

significant portion of each day within a scenario, such as if these activities comprise a user's day job. Naturally, it is not the case that instructors or researchers often have the time to spend several hours each day generating this information, much less over multiple variations of multiple scenarios.

1.4 Research objectives

At this point, we have clearly established a need for a more streamlined, reproducible method of developing scenarios for research and education. Simultaneously, this methodology should provide the variability and content needed for datasets to be interesting and useful in training students. These scenarios must also provide ground truth data to document the artifacts contained in each dataset, allowing users to determine its usefulness. In a classroom setting, this same information could be used to identify what should be contained in students' reports of the dataset.

Some research has been done into developing image synthesizers, which aim to automate part or all of dataset development, as described in chapter 2. This thesis aims to directly improve upon the foundations provided by past research, with the explicit goal of incorporating nearly all features from prior synthesizers. In particular, this thesis will describe the components and architecture necessary to build an effective forensic synthesizer for research and education with the following elements:

- The ability to create and export disk images, network captures, and volatile memory captures through arbitrary means;
- the ability to generate forensic artifacts in a modular manner, both with and without operating system virtualization;
- the ability to log every action in a standardized format, the Cyber-investigation Analysis Standard Expression (CASE) [48], to address existing dataset stan-

dardization concerns [34];

- and the use of modern Python libraries, standards, and practices, promoting future development by reducing the overall complexity of the framework and abstracting specific concepts where possible.

Furthermore, this thesis aims to leverage recent advancements in generative AI to streamline the development of complete forensic scenarios, as well as individual forensic artifacts to be added to a larger forensic dataset. Finally, this thesis will evaluate the viability of this framework in an actual classroom setting.

1.5 Contribution

This thesis contributes the **automated kinetic framework**, or AKF, a modern synthesizer framework that aims to provide the foundation of a larger forensic dataset ecosystem. This framework can be used to not only vastly reduce the time spent developing new datasets for research and education but also improve the discoverability of both AKF-generated and non-AKF-generated datasets. Additionally, by focusing on the long-term viability of AKF through its modular architecture, educators and researchers will be able to rapidly develop relevant datasets, even as new developments and advancements in technology occur.

In many ways, this thesis is intended to be the culmination of over 15 years of development in the field of synthesizers (beginning with Forensig2 [38]), re-implementing and adding to the unique contributions of previous synthesizers using modern techniques and technologies.

Chapter 2

Literature review

We begin with a literature review of two topics: publicly available forensic datasets and the contributions of prior synthesizers. More precisely, we explore the datasets that currently exist for digital forensics in greater detail, as well as gaps identified in these datasets by various authors. This will provide context for both the need for synthesizers and the gaps that these synthesizers have gradually filled.

2.1 Existing forensic corpora

Forensic datasets have been available for public use (sometimes by request) since the early days of the digital forensics field, though their sourcing and qualities have changed significantly over time. This topic was explored in far greater detail by Grajeda et al., who performed a survey of over 700 research articles to identify the various datasets used throughout the field. However, it is still important to highlight specific datasets relevant to the development of synthesizers.

Early datasets were primarily derived from real sources, whether made available to the public or otherwise. The earliest collections of real datasets include the used

hard drives collected by Garfinkel from 1998 to 2006 and the Enron email corpus obtained during the federal investigation of Enron [19]. Other early real datasets identified by Grajeda et al. include the public Apache mailing archive, the Reuters news corpora, and various facial recognition collections such as the MORPH corpus [49], all of which were made in the early to mid-2000s [21], [42].

A variety of synthetic datasets were also constructed during this early period. These datasets include the network captures obtained from simulated attacks conducted by the MIT Lincoln Laboratory from 1998 to 2000 [19], as well as standalone datasets used for tool validation as part of the early CFTT program developed by NIST. Other synthetic datasets during this period were generated as part of challenges, such as those produced for DFRWS conferences [44].

The variety of forensic datasets increased considerably towards the late 2000s, which can be credited to both the overall growth of the field (including the broader field of incident response) and computing as a whole. Various notable datasets described by Grajeda et al. include malware samples discovered "in the wild," natural language collections from multiple languages, and file-specific datasets such as collections of Microsoft Office files. It was also during this time that non-disk datasets, such as volatile memory dumps and network captures, became more prevalent. Although not explored by this thesis in detail, mobile datasets – such as smartphone disk images, mobile malware and applications, and SIM card images – also grew more prevalent.

Many of these datasets were not maintained as part of a larger collection with the explicit intent of providing them for digital forensics research. This began to change towards the late 2000s; for example, Garfinkel's collection of disk images would eventually evolve into the Real Data Corpus, growing to 30 terabytes by 2013 [42], [50]. The collection included disk images, flash drive images, and a variety of optical

discs sourced from real-world usage, requiring institutional review board approval to use. This collection would eventually be part of the Digital Corpora platform, which includes a set of purely synthetic datasets. Separately, NIST began developing the CFTT and CFReDS projects, both of which provide forensic datasets for various purposes. In 2021, Xu et al. compiled and published a repository of educational datasets that explicitly focuses on ease of use, realism, and breadth [51]. All corpora mentioned in this paragraph are currently actively maintained.

There are other datasets that are relatively unique and are often maintained as part of a larger niche collection. Collections of malware samples have grown significantly, in part because of the modern threat intelligence ecosystem supported by platforms such as VirusTotal. There exist datasets focusing on the dark web, such as those operating on the Tor network, including "black market" sites on which illegal goods are bought and sold. Finally, there are also network captures from various novel sources, such as iterations of the Collegiate Cyber Defense Competition and the networking infrastructure of university IT departments [21].

These datasets span a wide range of technologies – including different versions of the same technology – that require distinct methodologies to analyze effectively. However, the field faces the challenge of making datasets available that reflect current advancements in technology, such as new operating system versions or new applications. For example, instant messaging applications have changed considerably over the history of the field, ranging from MSN Messenger in the early 2000s to Skype, Discord, Telegram, Signal, Slack, and more. Artifacts from these applications must be handled differently, even if the value of the underlying service is essentially the same. Similarly, the strategies for analyzing Windows artifacts have changed significantly from version to version, as new registry keys become relevant in analyzing applications while others become unused.

This gradual "aging" of datasets, in which their relevance degrades over time, contributes to the continuous need for new datasets. Indeed, the need for new, novel datasets is one of the reasons identified by Grajeda et al. for the manual development of new datasets by research authors; it was often the case that a modern dataset simply did not exist for their needs. The other motivation is that the dataset used was never made public, either due to the author not having the resources to distribute the dataset themselves or because of legal and privacy concerns.

How are these datasets relevant to the development of synthesizers? The datasets throughout this section have demonstrated that many datasets have proved to be relevant in digital forensics research, even if not immediately evident. In turn, any effort to streamline the development of forensic datasets should be able to cover as many use cases as possible. Synthesizers must fulfill two requirements to achieve this:

- A synthesizer should be able to generate artifacts present in existing datasets, provided that the underlying technologies are still available.
- A synthesizer should be able to account for developments in operating systems, applications, or other technologies without requiring significant changes to the underlying architecture.

Indeed, the synthesizers described in the following section have explicitly addressed these two concerns. For example, many of these synthesizers have focused on implementing features that reflect the qualities of real-world datasets. These features include the ability to execute malware samples in a virtualized environment, send emails to arbitrary email servers, and insert data on removable drives – all of which generate forensic artifacts present in previously used datasets. Similarly, modern synthesizers are capable of generating disk images, network captures, and volatile memory dumps, in addition to extracting specific artifacts such as application-specific files. The gradual progression in the ability of synthesizers to generate specific arti-

facts is described in section 2.2, which explores specific contributions made by each synthesizer.

Of note is the generation of similar datasets that typically involve significant human interaction, such as public email distribution lists, photographs of human faces, and other transcripts of conversations. While prior synthesizers have not explored this in significant detail, it is explored as part of the generative AI work done as part of AKF in chapter 6.

The second issue, in which synthesizers must be extensible in such a way that they can support new applications, has been approached in several ways. This is described in greater detail in chapter 3, but has been a significant design consideration in the development of most synthesizers.

2.2 Analysis of existing synthesizers

Numerous frameworks have been built over the last two decades that aim to significantly reduce the effort involved in creating synthetic images from scratch by automating various application- and OS-specific actions according to provided instructions. (This is a subset of broader automation efforts throughout digital forensics, as described by Michelet et al. [52].) The functionality and availability of the frameworks described throughout the literature have varied considerably over time. However, the goal of these frameworks has largely remained consistent: they all aim to provide a rapid method for instructors to develop forensic labs for students. Note that low-level implementation details are described in later, more relevant sections throughout this thesis for clarity rather than providing detailed analyses as part of this chapter. Before analyzing and adapting the low-level implementations of prior synthesizers, it is best to first contextualize their broader contributions.

The first identified effort to streamline the creation of forensic labs through a high-level language was published by Adelstein et al. in 2005 through the development of **FALCON**, the Framework of Laboratory Exercises Conducted Over Networks [31]. It proposed an architecture for automatically creating lab exercises, deploying them to a virtual lab, and evaluating students' actions in the lab compared to the ground truth. Very few implementation details are provided in the paper, though some examples of its usage are included. Although not a synthesizer, related concepts can be seen in the description of **CYDEST**, the CYber DEfenSe Trainer [24], which similarly provided complex virtualized lab environments to students via the internet. Again, the authors provide limited implementation details, and it is unclear if either FALCON or CYDEST are publicly available or actively maintained.

Forensig2, described by Moch and Freiling in 2009 and revisited in 2012, appears to be the first detailed description of an image synthesizer [38], [53]. Users define scenarios through a Python 2 library that provides abstractions around various virtual machine operations, such as formatting disks, creating partitions, and copying files from the host to the virtual machine. Actions are performed live through a Qemu-based VM, eventually producing a ground truth report and (effectively) an image to be analyzed by students. The source code for Forensig2 is not currently maintained and appears to be unavailable.

The **Digital Forensic Evaluation Test (D-FET)** platform described by William et al. in 2011 provides a custom scripting language for users to define system- and user-level actions [54]. Similar to FALCON and CYDEST, it provided virtualized labs through VMware ESXi. Unlike most other frameworks, it was primarily built to provide a platform to efficiently evaluate digital forensic tools on a scalable infrastructure rather than streamline the process of building and distributing labs for instructors (though it is mentioned that the infrastructure was also used to host student labs).

It is unclear if a public implementation is available.

Russell et al. define an XML specification, the **Summarized Forensic XML (SFX)** language, to describe scenarios as a sequence of various actions on various partitions of a disk [46]. This can be passed into an interpreter to produce a disk image that can be analyzed by students. Besides high-level operations such as disk partitioning and file copying, it also provides Windows- and Linux-specific routines for minimizing partition sizes to reduce the size of files irrelevant to a scenario, often those associated with the OS. It also briefly describes approaches for updating web browsers and the Windows registry to reflect certain actions. Limited implementation details are provided, and it is unclear if a public implementation is available.

Yannikos et al. define a framework for describing scenarios as a series of Markov chains. The framework allows users to visually define scenarios through a graph view in which individual nodes represent distinct actions to be taken by the synthesizer, with probabilistic transitions deciding which node to execute next [42]. Its focus on non-determinism makes it well-suited for quickly developing variations of the same scenario, though it is unclear what application- or OS-specific routines are provided.

The remaining frameworks are broadly similar in that they are all implemented with Python, providing users with a Python library to define and generate scenarios. Various functions in the library provide abstractions to complex actions, such as Facebook browser activity or sending emails on the Thunderbird application. Additionally, each framework produces some form of detailed output that can be used as answer keys or ground truth in an instructional setting. A brief summary of the notable aspects of each of these frameworks relative to each other is provided below:

- **ForGeOSI** [55] introduced the use of the VirtualBox SDK to automate various operations, forming the basis for much of the work done as part of VMPOP.
- **ForGe** [56] is specifically designed to generate NTFS and FAT32 images, fo-

cusing on placing data directly onto disk images without a virtual machine by maintaining and serializing custom data structures for supported filesystems.

- **EviPlant** [47] encompasses a novel method for efficiently distributing generated disk images, which is achieved by generating and distributing differential "evidence packages" to apply to a base image file. An OS-specific injection tool creates relevant artifacts according to the evidence package. Since the base image must only be downloaded once, each reconstructed disk image is significantly smaller than if distributed as standalone images.
- **VMPOP** [33] provides an architecture for elaborate VirtualBox control (such as attaching USB devices and starting video captures) in addition to various OS-specific commands. Provided routines for Windows include creating restore points, installing programs, mapping network drives, setting registry values, and more. Although the architecture as a whole is platform-independent, the provided implementations operate with VirtualBox and Windows. **TraceGen** [57] is similar in that it makes use of the VirtualBox API for various operations but is more like hystck in its use of a Python-based agent to carry out application-specific actions.
- **hystck** [58] provides routines for automating OS- and application-specific commands through both YAML configuration files (passed through an intermediate interpreter script) and/or Python scripts (executed normally). The framework produces network captures and disk images; similar to EviPlant, it supports "differential" images that can be distributed and applied to "template" images.
- **ForTrace** [45] is the most recently developed synthesizer, which directly builds upon hystck by providing volatile memory captures (alongside disk and network captures) in addition to various other new features (such as the ability to execute PowerShell scripts to create Windows artifacts), with a focus on a modular

architecture. A variant focusing on Android artifact generation was developed in 2024 [59].

Clearly, a considerable amount of work has been done to streamline the process of developing images, although the availability and functionality of each framework vary greatly. Notably, with the exception of ForTrace, none of these works are direct extensions of past works, implying that at a fundamental level, it was necessary for new frameworks to be developed from scratch to support the framework authors' needs. This is not directly stated in any of these works, with the exception of hystck [58]. However, it can be reasonably concluded that the lack of maturity, availability, and maintenance of prior works contributed to the independent development of the other frameworks. These issues – and potential solutions – will be addressed in more detail in chapter 3.

Chapter 3

Architecture and design

3.1 Motivation

As described previously in section 2.2, with the notable exception of ForTrace [45] and its related synthesizers, no synthesizer has been an extension of another synthesizer. This raises the question – why reinvent the wheel by developing yet another distinct architecture for this thesis? This section briefly explores the deficiencies in existing synthesizers and explains why these are issues that warrant building a new architecture from scratch rather than extending an existing synthesizer.

The motivations for developing entirely new codebases instead of extending existing synthesizers have varied considerably. One reason is that several synthesizers are not open source and thus cannot easily be extended, as is the case with Forensig2 [38] and TraceGen [57]. Another reason is that the focus of certain synthesizers results in an architecture that is simply incompatible with the goals of newer works. For example, ForGe's architecture [56] focuses largely on direct filesystem manipulation to generate forensic artifacts and is unsuitable for a synthesizer requiring virtualization.

Similarly, synthesizers such as VMPOP [33], which exclusively leverage agentless artifact generation as described in section 4.1, require significant architectural changes to support agent-based artifact generation.

However, perhaps the largest motivation for constructing new synthesizers is the lack of ongoing support for virtually all synthesizers. It appears that no synthesizer has gained significant traction within the broader forensic community, possibly with the exception of ForTrace; for the synthesizers that *are* open source, none are under active development and maintenance. Additionally, the forensic datasets generated by these synthesizers have not seen significant adoption in either education or research; many instructors continue to use the human-generated datasets available on public platforms.

The inflexibility of prior synthesizers, combined with the overall lack of support and success of synthesizer-based datasets, contributes to the disparate codebases that are now observed today. However, this is not to say that the individual contributions of each prior synthesizer cannot be merged into a single project that resolves many of the architectural barriers that have limited the adoption and extension of existing synthesizers.

In turn, AKF is built on the following four pillars to help promote its long-term usage. In particular, these design principles allow it to generate datasets that address several of the considerations raised by Horsman and Lyle in the construction of various datasets, such as the need for comprehensive documentation, an awareness of the "realism" of the resulting dataset, and transparency in the scenario development process [34].

First, AKF conforms to modern Python development practices. This includes the use of modern project management practices (such as the use of `uv` [60] and `pyproject.toml`, rather than the use of `setup.py` observed in older synthesizers), static linters

(`flake8` [61]), and style enforcers (`black` [62] and `isort` [63]) to promote adherence to the PEP8 standard. Additionally, AKF's libraries make heavy use of Python 3.11+ features, including type hinting and special type annotations, which allow for tools such as `mypy` [64] to perform static type checking. In addition to significantly improving the development experience, these practices also increase the likelihood of discovering errors earlier in the development process.

Second, AKF takes a modular, agent-based approach to implementing application-specific functionality. This allows AKF to use existing automation frameworks for web browsers and other applications, greatly simplifying the codebase when compared to the implementation of the same features in other synthesizers. This focus on flexibility makes it significantly easier to install the agent, implement new application-specific features, and more. This is a critical design focus, given AKF's dependence on agents for most of its application-specific functionality.

Third, AKF is architected to maintain feature parity with most prior synthesizers. That is, although AKF deviates considerably from the implementation details of prior synthesizers, this does not come at the loss of prior advancements in the field. In particular, AKF supports all three artifact generation techniques described in chapter 4 using hypervisor-agnostic interfaces, reducing the tight coupling that made certain features challenging to implement across different synthesizers. This reduces the likelihood that a new feature or technique will require a significant architectural change to support it.

Finally, AKF is designed with the explicit intent of developing an ecosystem of AKF-generated datasets, promoting long-term usage. This is reflected in the design of AKF's logging and reporting mechanisms as described in chapter 5; the use of an RDF-based standard, CASE, allows for arbitrarily complex queries against AKF datasets. These queries can be made in bulk, significantly improving the ability of

researchers to find and use datasets that may be relevant to them.

These four pillars are reflected throughout the design of AKF's architecture, which is described in the following section and the following three chapters of this thesis.

3.2 The AKF architecture

At a minimum, every synthesizer must fulfill these high-level requirements through some means, derived mainly from the criteria developed by Horsman and Lyle [34]:

- The synthesizer must accept commands on how it should operate. These commands should serve as a form of self-documentation, in which it is clear to another person *what* the expected contents of the image are, as well as the *intent* behind these commands.
- The synthesizer must be able to accept external inputs, such as files and other binary data, to be used as part of artifact and dataset generation.
- The synthesizer must implement the mechanisms and technologies necessary to carry out the commands it has been provided – that is, it must be able to generate artifacts. Such mechanisms should be independently verifiable and leave as little extraneous information as possible.
- The synthesizer must be able to generate a final output (such as a disk image), along with ground truth and reporting that describes the expected contents of that image. This should include a well-structured description of the dataset, a unique identifier, and explicit labels applied to data of "evidential value" where possible.

These four requirements are fulfilled by various mechanisms throughout AKF. Detailed architectural diagrams can be seen in Appendix A, but a simplified diagram of AKF's top-level "modules" is shown in Figure 3.1.

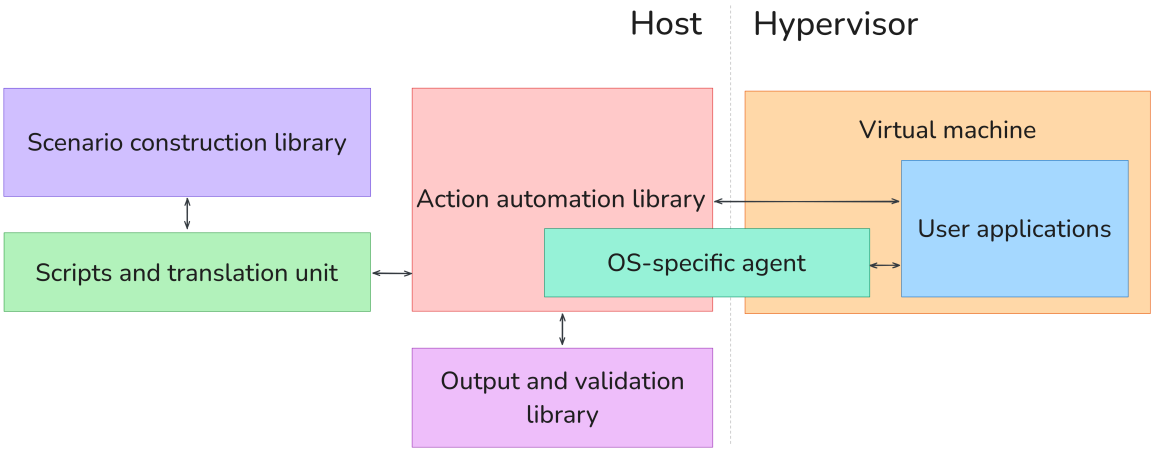

Figure 3.1: Simplified AKF architecture diagram

AKF's seven top-level modules can be grouped into three distinct concepts, each covering a specific chapter.

The first set of modules is responsible for artifact generation. This encompasses three major systems - a hypervisor and its associated SDK, an OS-specific agent, and `akflib`. `akflib` is a Python library containing the abstract interfaces and concrete implementations necessary to generate artifacts and datasets. This includes routines for directly interacting with hypervisors, issuing commands to virtual machines, and directly modifying disk images. This library is also the foundation for OS-specific agents, which carry out actions on the virtual machine on behalf of the host. These modules are described in greater detail in chapter 4 and correspond to the *action automation library*, the *OS-specific agent*, and the *virtual machine* (as well as the *user applications* running on the machine).

The second set of modules is responsible for logging and reporting. This encompasses independent libraries for generating outputs and ground truth, as well as the various logging-related mechanisms contained throughout the artifact generation libraries. These modules are responsible for exporting and documenting artifacts generated by AKF; in particular, they make heavy use of CASE, a standardized ontology

for documenting the contents of forensic datasets [48]. This is covered in chapter 5 and corresponds to the *output and validation library* and any associated components within the *action automation library*.

The final set of modules is responsible for invoking AKF itself and supporting scenario development. AKF is an *imperative* synthesizer, which means that commands are written and executed using an imperative language (Python 3) that dictates precisely *how* scenarios should be constructed. However, AKF also supports a *declarative* syntax, which allows users to specify *what* forensic artifacts and datasets are generated without the need to understand the AKF libraries and write Python code. Additionally, AKF contains generative AI tools for constructing scenarios in the declarative syntax as well as individual artifacts. This is discussed in chapter 6 and covers the *scenario construction library*, the *translation unit*, and any scripts that leverage AKF libraries.

The following three chapters will focus on these module groups. In simpler terms, this thesis addresses the following questions in order:

- How do we automate or streamline the generation of artifacts?
- How do we document and report on the artifacts and datasets that are generated?
- Given the solutions that address these two questions, how do we actually use them to build scenarios?

Chapter 4

Action automation

At the core of every synthesizer is the ability to generate forensic artifacts as part of a larger scenario; this chapter addresses the AKF modules responsible for automating artifact generation, depicted in Figure 4.1 below. The detailed diagrams for these modules are part of Appendix A, depicted in Figure A.2 and Figure A.3.

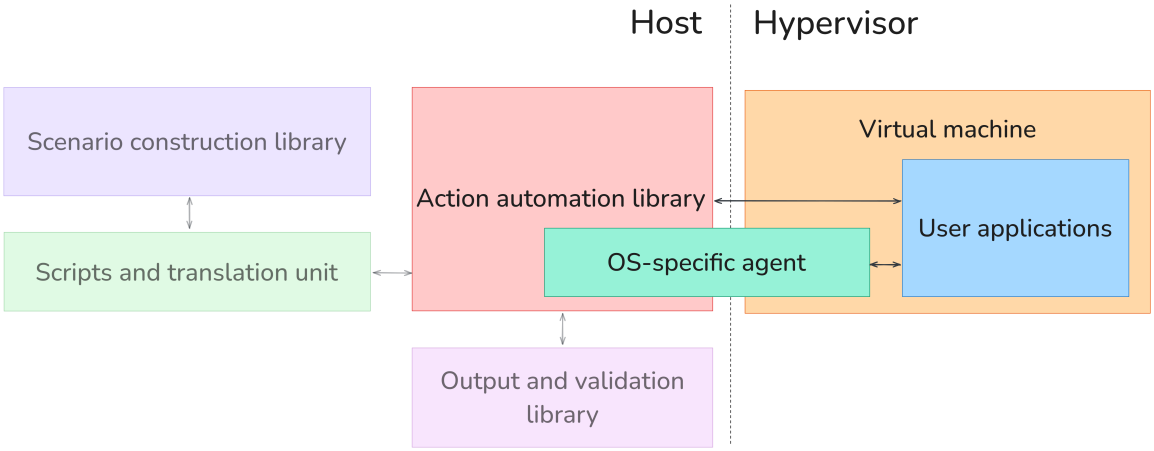

Figure 4.1: Simplified AKF architecture diagram for scenario construction modules

Each of the synthesizers described in section 2.2 takes one of three approaches to artifact generation, as partly described by Scanlon et al. [47]:

- **Physical**: No virtualization of software or hardware ever occurs; data is written

directly to the target medium, such as a disk image or virtual hard drive.

- **Agentless logical**: The synthesizer interacts with a live VM to generate artifacts. Interaction is achieved without the need for custom software to be installed on the VM; instead, actions are achieved using the hypervisor itself or a remote management tool native to the virtualized operating system.
- **Agent-based logical**: The synthesizer interacts with a dedicated client, or agent, on a live VM to carry out actions. The VM must have the agent installed before any interaction can occur.

These three approaches are not mutually exclusive within a single synthesizer, though many prior synthesizers have used only one approach to generate artifacts. **??** below denotes the approaches used by each of the synthesizers previously discussed. Where source code is unavailable, a best effort was made to identify the approach used by a particular synthesizer based on its published paper, if one exists; otherwise, the entire row contains question marks.

Synthesizer	Physical	Agentless	Agent-based
FALCON [31], 2005	?	?	?
CYDEST [24], 2008	?	?	?
Forensig2 [38], [53], 2009	Yes (filesystem mounting; filesystem-independent editing)	Yes (over SSH only)	No

Synthesizer	Physical	Agentless	Agent-based
D-FET [54], 2011	Yes (filesystem mounting; filesystem-independent editing)	No	No
SFX [46], 2012	Yes (filesystem mounting)	No	No
Yannikos et al. [42], 2014	?	?	?
ForGeOSI [55], 2014	No	Yes (hypervisor interfaces)	No
ForGe [56], 2015	Yes (filesystem-aware editing)	No	No
ForGen [65], 2016	No	No	No
EviPlant [47], 2017	Yes (filesystem-independent editing)	No	Yes (unknown mechanism)
VMPOP [33], 2018	No	Yes (hypervisor interfaces)	No
hystck [58], 2020	No	No	Yes (Python agent)

Synthesizer	Physical	Agentless	Agent-based
TraceGen [57], 2021	No	No	Yes (unknown mechanism)
ForTrace [45], 2022	No	No	Yes (Python agent)

There are advantages and disadvantages to each approach, in addition to requiring distinct implementation techniques for each. The remainder of this chapter analyzes each of these three approaches in greater detail, describing the implementation details of prior synthesizers and comparing them to those of AKF's.

More specifically, this chapter addresses the functionality of the action automation library (`akflib`) to generate artifacts by either interacting with a live virtual machine or by directly editing disk images stored on the host. We explore how logical artifacts are generated at a lower level using hypervisor APIs and an OS-specific agent running on the virtual machine. We conclude by describing the generation of physical artifacts through direct filesystem and disk image editing.

4.1 Agentless artifact generation

Agentless artifact creation describes one of two general techniques. The first is emulating "normal" human interaction by leveraging virtualized human interfaces – such as the monitor, keyboard, and mouse – to directly manipulate a GUI-based

operating system. The second is using (remote) management utilities included with the operating system, typically an interactive shell.

AKF allows users to perform agentless artifact creation through a hypervisor-agnostic interface; a concrete implementation of this interface using VirtualBox is provided. This VirtualBox-specific functionality is derived mainly from ForGeOSI [55], which uses the VirtualBox SDK and a Python implementation of the VirtualBox COM API [66] to carry out the vast majority of its tasks. This was adapted in VMPOP [33], which also introduced the notion of a generic hypervisor interface that allows for synthesizer routines to use arbitrary hypervisors so long as required functionality is implemented.

Note that the use of hypervisor-specific guest software, such as VirtualBox Guest Additions and VMWare Tools, is treated as an agentless approach for this thesis. Although this inherently requires installing "unusual" software on the virtual machine, it is very distinct from typical user software. In many cases, artifacts generated by hypervisor guest software can be identified, isolated, and ignored.

Relevant AKF submodules for agentless generation are depicted as opaque elements in Figure 4.2.

4.1.1 Human interfaces

The use of human input devices – the keyboard and mouse – is the primary approach taken by VMPOP [33] and ForGeOSI [55] to generate artifacts. For example, VMPOP uses a sequence of keyboard strokes to focus and interact with UI elements, such as clearing User Account Control dialogs on Windows or starting applications with Win+R. It achieves this by making calls to the VirtualBox API, which issues scancodes to the virtual machine.

This approach most accurately reflects how an actual human would interact with

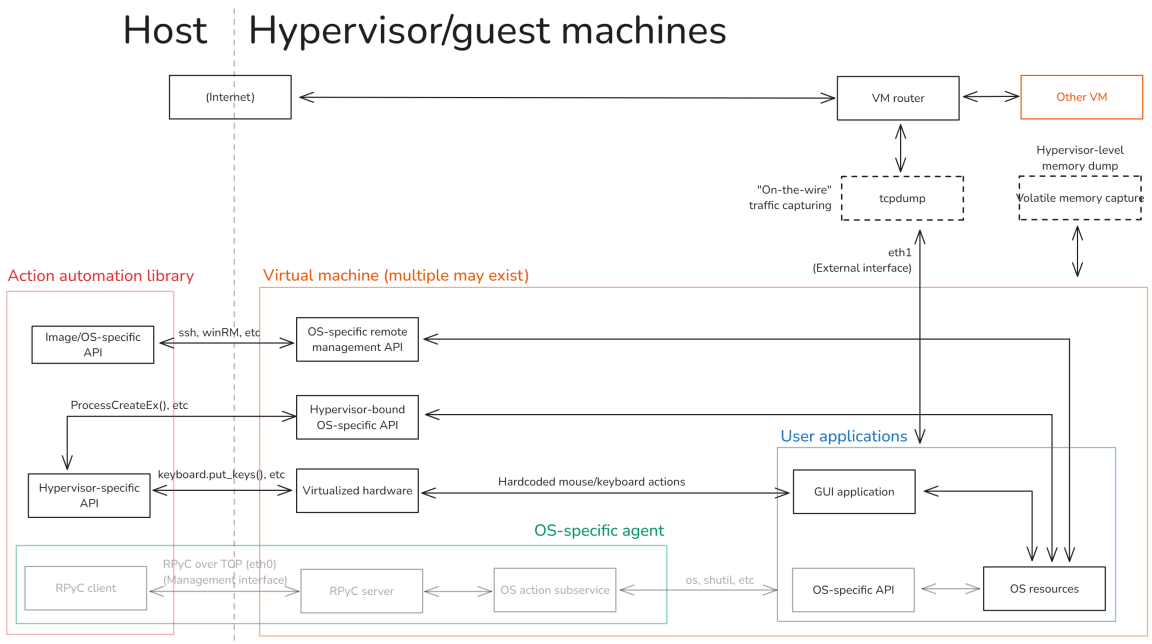

Figure 4.2: Abridged submodule diagram for agentless artifact creation

a machine. As a result, this greatly reduces the **Synthesis pollution** that occurs as a result of generating artifacts. However, this often leads to verbose scripts that are only capable of performing very specific actions. The keyboard and mouse actions required to fulfill a particular action can change significantly between versions of the same application, versions of the same operating system, and varying screen sizes. For example, VMPOP handles User Account Control (UAC) prompts on machines prior to Windows 7 by sending an Alt+Tab keystroke to the machine, clicking the mouse at the center of the screen to focus the UAC prompt, and then sending Alt+C to accept the prompt. On machines running Windows 7 or later, this is instead achieved by focusing the UAC prompt with the mouse and then sending Alt+Y to accept the prompt.

Similarly, VMPOP interacts with browsers exclusively through the use of keyboard shortcuts. VMPOP supports simple interactions with a small number of websites, such as logging into Microsoft or Google accounts, but is dependent on the form

elements remaining the same over time. That is, VMPOP does not inspect the contents of the current webpage to perform actions; it cannot react to design changes. If the page has new focusable elements, the same keystroke sequence may no longer achieve the desired effects. This lack of runtime logic, which amounts to operating a computer with the monitor off, leads to brittle scripts that can be tedious to fix and update.

AKF's VirtualBox implementation allows the user to issue mouse events at absolute coordinates, though it does not support general mouse movement, such as clicking and dragging. It also allows the user to issue a sequence of press-and-release events while holding down specified keys. While this is theoretically sufficient to emulate nearly all actions that a human would normally perform with a mouse and keyboard, the AKF agent (as described in subsection 4.2.2) also exposes a more flexible mouse and keyboard automation API. More generally, the AKF agent leverages automation frameworks capable of runtime UI analysis, reducing the inflexibility of application-specific scripts.

TraceGen [57] notes that the ideal future is to use computer vision and AI to automate user actions from high-level prompts. Performing a Google search in a specific browser would currently require a sequence of predefined keystrokes, assuming an agentless approach is needed. While this is explored in greater detail in section 8.1, recent advancements in LLMs may make it possible to allow a machine to perform arbitrarily complex tasks on a GUI-based operating system using natural language – an approach that can be integrated into AKF in the future.

4.1.2 Management utilities

The alternative is to use existing management utilities, typically a shell, which are native to the virtualized operating system and are capable of carrying out commands.

This approach can also be divided into two categories: local management utilities, such as Bash and PowerShell, and remote management utilities, such as SSH and WinRM.

Local management utilities typically refer to scripting languages that are available as part of the operating system and can be used to manage most or all operating system resources. For example, PowerShell allows users to modify registry keys, invoke applications, create users, and more. Similarly, any standard Linux shell, such as Bash or Zsh, can be used to install packages and run various command-line applications. These shells can be invoked by either opening and focusing a terminal window (for example, through the Win+R shortcut on Windows) or directly executing scripts through hypervisor guest additions.

In particular, VMPOP [33] makes heavy use of local management utilities; it implements much of its functionality through a collection of PowerShell and batch scripts. For example, VMPOP allows users to focus a window by process ID, process name, or window title. This is achieved by getting a PowerShell handle to the process using `Get-Process`, using `Add-Type` to add a local C# function that is capable of sending keyboard events through `user32.dll`, and then holding the Alt key while using `AppActivate` to focus the window and bring it to the foreground. VMPOP leverages similar scripts for launching and terminating processes by name, uninstalling programs, creating a Windows restore point, and more.

Similarly, remote management utilities allow a remote device to invoke local management utilities, typically over an SSH server or WinRM. In most synthesizer architectures, remote management utilities require that the host and guest machines can communicate with each other. This is typically achieved through a NAT or host-only interface managed by the hypervisor. This approach is taken by Forensig2 [38], which exclusively connects to a running SSH server on a virtual machine to carry out user

actions.

The use of remote management utilities to automate actions typically undertaken by users is not uncommon, especially with the prevalence of infrastructure as code solutions. For example, the open-source Ansible framework simplifies the configuration of Windows and Linux devices to simple, YAML-based files called "playbooks." These playbooks contain a sequence of high-level tasks to perform, such as installing packages, managing local user accounts, and running scripts, which are executed on multiple remote machines in parallel. (This high-level scripting language is the inspiration for AKF's high-level scripting language, described in section 6.3.)

It is worth noting that although this approach does not require installing new software on the machine, most operating systems perform some degree of logging when their management utilities are invoked. For example, the `sshd` daemon logs connections regardless of whether a NAT or host-only network interface is used. This behavior may make it difficult to separate or remove logs unrelated to a scenario. Similarly, the invocation of PowerShell scripts generates event logs, which can be particularly noisy if Script Block Logging (which logs the execution and content of all PowerShell scripts) is enabled.

AKF inherits the capabilities of VMPOP, allowing users to leverage the VirtualBox API and Guest Additions to execute arbitrary processes. For example, users can invoke scripts to automate OS-specific configuration, such as using PowerShell to set registry keys. However, AKF does not currently implement any agentless OS- or application-specific functionality through this method; many of the actions and artifacts previously generated through OS-specific scripts in VMPOP [33] and other synthesizers can be implemented with greater flexibility through the agent. How this flexibility is achieved, as well as why it is the preferred method for implementing application-specific functionality, is described in the following section.

4.2 Agent-based artifact generation

Agent-based artifact creation involves the use of a dedicated program on the VM that serves as an interface between the host machine and the guest machine. This program runs commands natively on the virtual machine on behalf of the host machine, accepting commands over a dedicated network interface. This allows for greater flexibility and more complex actions to be taken when compared to agentless approaches. In particular, it allows application-specific functionality to be implemented using existing automation frameworks such as Selenium [67], Playwright [68], and PyAutoGUI [69]. However, this approach often leads to **synthesis pollution**; besides the presence of software that would never exist on a typical user's machine, agents often do not interact with applications the same way that a human would. As a result, the synthesizer should document known pollution that can be ignored for educational and testing purposes.

Relevant AKF submodules for agent-based generation are depicted as opaque elements in Figure 4.3.

4.2.1 Analysis of the ForTrace agent

Agent-based artifact creation is the approach taken by hystck/ForTrace [45], [58], which refers to its agent as an "interaction manager." Because ForTrace provides extensive agent functionality and is by far the most mature synthesizer, AKF's agents borrow heavily from ForTrace's approach. We briefly describe ForTrace's implementation of agents here, with a more detailed analysis in section B.2.

ForTrace agents are simply an entire copy of the ForTrace codebase copied over to the virtual machine – that is, the "agent" and "server" share the same codebase but have different entry points and use different parts of the ForTrace library. Com-

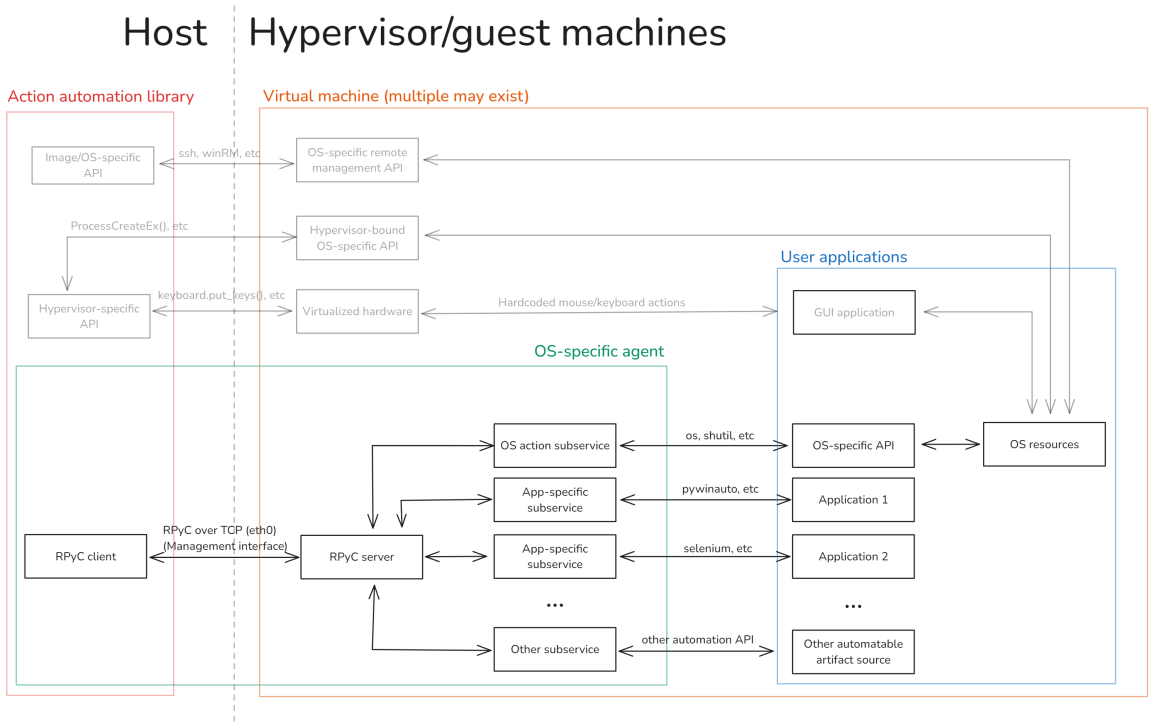

Figure 4.3: Abridged submodule diagram for agent-based artifact creation

munication between the agent and the server occurs over a dedicated TCP socket using a simple ASCII-based protocol; to execute commands, the server sends a space-delimited string containing the application, function, and arguments to run. Results are similarly returned by the agent as structured ASCII messages. These messages are typically sent over a dedicated "management" network interface to exclude them from network captures for scenario-related activities.

Application-specific functionality for agents is organized into individual files (Python modules), where each file includes a set of commands or actions for a single application. These files contain both server-side code (the logic for *constructing* the ASCII message) and agent-side code (the logic for *interpreting* the message and *executing* the command) as a set of classes with a common prefix. This common prefix is used to automatically discover the corresponding agent-side code given a command issued by the server-side code.

Two key implementation details should be noted here. First, commands are sent through a simple string-based protocol. This simplicity makes it easy to debug issues arising from the protocol itself, but is relatively inflexible as a result. In particular, it is challenging to send complex Python objects as arguments or return values since these objects often cannot be easily serialized to a string without loss of information. An example relevant to AKF is passing Playwright browser objects, which contain an internal state that is difficult to extract and reconstruct using strings alone.

The second implementation detail involves the discovery and execution of commands indicated by the protocol. ForTrace depends on Python's runtime introspection to discover the correct modules and functions to call based on the contents of a command string, importing these modules during runtime. While this is a valid approach, it is more complex (and difficult to follow) than the approach taken by AKF.

4.2.2 The AKF agent

AKF borrows heavily from the Python agent-based approach of ForTrace but improves upon its architecture in several respects. Details of low-level improvements over ForTrace's agent architecture are described in more detail in section B.2.

Perhaps the most significant difference is the use of RPyC, a library for symmetric remote procedure calls, for agent communication [70]. Although the RPyC protocol is symmetric, it is often used in typical client-server architectures to allow clients to manipulate remote Python objects as if they were local objects, as well as invoke remote (server) functions using local (client) parameters. Delegating the serialization and deserialization of complex objects to RPyC allows us to perform complex operations that would have been difficult to implement with the simple string-based protocol of ForTrace.

In “new-style” RPyC, this is achieved by running a *service* on the device where remote operations should be performed. Services expose a set of functions and attributes that may be accessed remotely by an RPyC client, listening on a specified TCP port for requests to access these exposed elements. Clients access these functions and attributes by name as if they were local objects; arguments passed to functions are serialized and deserialized in the background, as are the results of function calls and attribute accesses.

AKF’s application-specific functionality is divided into individual RPyC “subservices” created on demand. The agent’s main loop is itself a “root” RPyC service that is responsible for creating and destroying these subservices upon request; all subservices are known to the root service at initialization, eliminating the need to perform runtime introspection to find application-specific modules. These subservices are analogous to the agent-side code of individual ForTrace modules, interpreting arguments and executing commands on behalf of the host machine.

Clients are largely decoupled from the server’s implementation of individual services; they do not directly call or import RPyC service functions. Instead, they call these functions by name from an untyped connection object. Because the exposed functions (as well as their signatures) and attributes cannot be inferred from the raw RPyC connection alone, clients can implement a typed, concrete API to construct RPyC calls. This abstracts the raw RPyC call (and the existence of an RPyC connection) away from the user while also providing the signatures of remote functions where there would otherwise be none. This allows for type checking and autocompletion during development. This is analogous to the client-side code of individual ForTrace modules, constructing messages to the agent to execute commands.

Additionally, because the RPyC service and the API to that service are separable, this allows us to break the API (which is used in scenario scripts) and the agent logic

itself into two separate libraries. This has two advantages – it makes it significantly easier to build and generate standalone executables using tools like PyInstaller [71], and it may also slightly reduce the size of the agent when installed onto the virtual machine. Unlike ForTrace, whose agent installation process requires a batch script installing various libraries and Python through the Chocolatey package manager, AKF's agent only requires that a single executable is copied over and configured to run on startup (in addition to setting relevant firewall rules). The agent setup process can be further simplified using Vagrant, which is described in section 6.2.

From an implementation and usability perspective, this design provides three significant improvements over the ForTrace protocol. First, the routing of functions is wholly delegated to RPyC. Instead of manually constructing a message with the function name and its associated parameters (as strings) over the network, the process of serializing parameters and routing them to the correct function call is abstracted away by RPyC.

Second, this allows us to pass and return arbitrarily complex objects (for which we do not have to manually write the serialization and deserialization logic). When passing complex objects from the agent to the server or vice versa, a reference to the object is sent over the network and wrapped by a *proxy object*, which behaves like the original object [72]. Importantly, it is not necessary to distinguish between local and remote/proxy objects of the same type when writing code, which eliminates the extra complexity of using proxies.

Finally, the ability to interact with complex remote objects allows us to significantly reduce the actual code written as part of the API exposed to the host. For example, there is no need to implement a wrapper for every method available as part of a Playwright page object; instead, a reference to the Playwright object *running on the virtual machine* can be given to the host machine. Instead of writing individual

methods for opening pages, navigating to specific elements, and so on, we can simply use the methods that already exist in the Playwright object – any local calls on the host's proxy object will lead to remote outcomes on the host, as desired. This, of course, does not preclude the ability to write convenience methods for more complex actions requiring the Playwright object.

The list of applications supported by the AKF Windows agent is described in **??** below.

Module	Dependencies	Details
ChromiumPlaywright	Playwright [68]	Allows arbitrary webpages to be visited on Chrome and Edge, as well as perform complex actions such as completing forms and clicking links based on HTML selectors

The generic hypervisor interface is used to support agent discovery and communication. To avoid polluting network captures with agent-related packets, virtual machines are expected to use a NAT adapter for Internet communications and a "maintenance" host-only adapter for agent-specific communications. In turn, hypervisor-specific implementations must expose the ability to discover the IP address of the host-only adapter. This allows AKF scripts to communicate with the root RPyC service and any subservices over the host-only adapter, concealing them from network captures on the NAT adapter.

4.3 Physical artifact generation

4.3.1 Background

Physical artifact creation encompasses any technique in which the virtualization of an operating system is not used to generate artifacts. This allows the synthesizer to bypass the operating system or related software that could lead to undesirable non-deterministic behavior. This is sometimes called *simulating* the creation of artifacts rather than *virtualizing* their creation.

For example, a scenario developer may want to guarantee that a particular deleted file is partially overwritten by another file, ensuring that the deleted file is recoverable from the slack space of the newly placed file. However, it is extremely difficult to force the reuse of the same physical disk clusters from a userspace application. Operating systems rarely expose low-level filesystem functionality to applications; furthermore, operating systems are still subject to hardware drivers that regularly rearrange physical space, such as those that engage in wear leveling.

While it may be possible to predict the outcome of these wear leveling techniques, as demonstrated by Neyaz et al., this is far from the determinism necessary for research and tool validation [73]. Additionally, background programs and services may perform file operations that are difficult to predict. Finally, there is hardware-related non-determinism, such as that caused by faulty hardware or cosmic rays. (There is ongoing work in building deterministic platforms, such as the deterministic hypervisor developed by Antithesis [74]. Such platforms are outside the scope of this thesis but could be integrated into synthesizers in the future.)

In turn, it is sometimes necessary to bypass the operating system to reliably place data on a disk. Two primary strategies are used to achieve physical artifact creation.

The first strategy is mounting the filesystem and performing typical read/write

operations. Filesystem mounting is used by several synthesizers, including Forensig2 [38] and SFX [46]. For both of these synthesizers, physical artifact creation is achieved by simply allowing the user to specify a file to copy from the host machine to a specific file path on the disk image.

The second strategy is performing writes directly to the underlying disk image, bypassing the filesystem entirely. For example, ForGe [56] maintains a virtual representation of supported filesystems, using its own data structures to represent a FAT32/NTFS filesystem. This allows it to quickly identify and place data in unallocated or slack space. In contrast, synthesizers such as EviPlant [47] simply allow the user to provide an offset into the disk image where arbitrary data will be written. Both techniques involve directly writing to specific physical addresses, with the only distinction being filesystem awareness.

It is theoretically possible to construct complete forensic datasets through simulation alone. If a user knows what artifacts are generated through the execution of an application and how these artifacts are generated, it may be possible to artificially generate these artifacts as if the application had actually been running. An example may be editing the history database of a particular browser contained on a filesystem by mounting it, parsing it, and adding new entries; this would make it appear as if the machine was used to browse these websites without requiring virtualization.

However, this technique is challenging to implement in practice. For example, a scenario developer would need to ensure that the artifact is consistent with other artifacts on disk, since typical Windows application usage would generate relevant artifacts in prefetch files and jump lists. Additionally, physical artifact creation does not lend itself well to generating network captures and volatile memory. Because no operating system is virtualized, it is naturally the case that there are no applications making network requests or using volatile memory as part of execution. Building a

network capture or volatile memory dump from scratch is difficult, at best.

It is important to note that most artifacts generated by physical techniques can still be created non-deterministically through logical means. A user could create files of interest, delete them, and then write new files to disk through a virtual machine, possibly leading to the same outcome as simply writing these files directly to the filesystem without virtualization. As described previously, there is no guarantee that the new files will occupy the same physical clusters as the deleted files. However, this may be acceptable, such as a scenario in which an instructor merely wants to demonstrate examples of what could occur when deleting a file. If there is no need to carve deleted files from slack or unallocated space reliably, then logical approaches are sufficient.

4.3.2 AKF implementation

As described in the prior section, there are three techniques for physical artifact planting implemented by prior synthesizers. These are:

- **Filesystem mounting**, as done by Forensig2 [38] and SFX [46], in which the filesystem is mounted to the host and edited directly.
- **Filesystem-independent direct editing**, as done by EviPlant [47], in which edits to specific physical addresses on the disk image are made without any parsing or knowledge of the underlying filesystem.
- **Filesystem-aware direct editing**, as done by ForGe [56] and EviPlant [47], in which filesystem data structures are parsed to determine the physical address(es) of the disk image to write to. (How EviPlant achieves this is not known.)

AKF supports all three to varying degrees, with significant improvements in

filesystem-aware editing over prior synthesizers. All three techniques are implemented in the opaque submodules indicated in Figure 4.4.

!39.4 - Action automation 2025-03-11 01.28.06.excalidraw

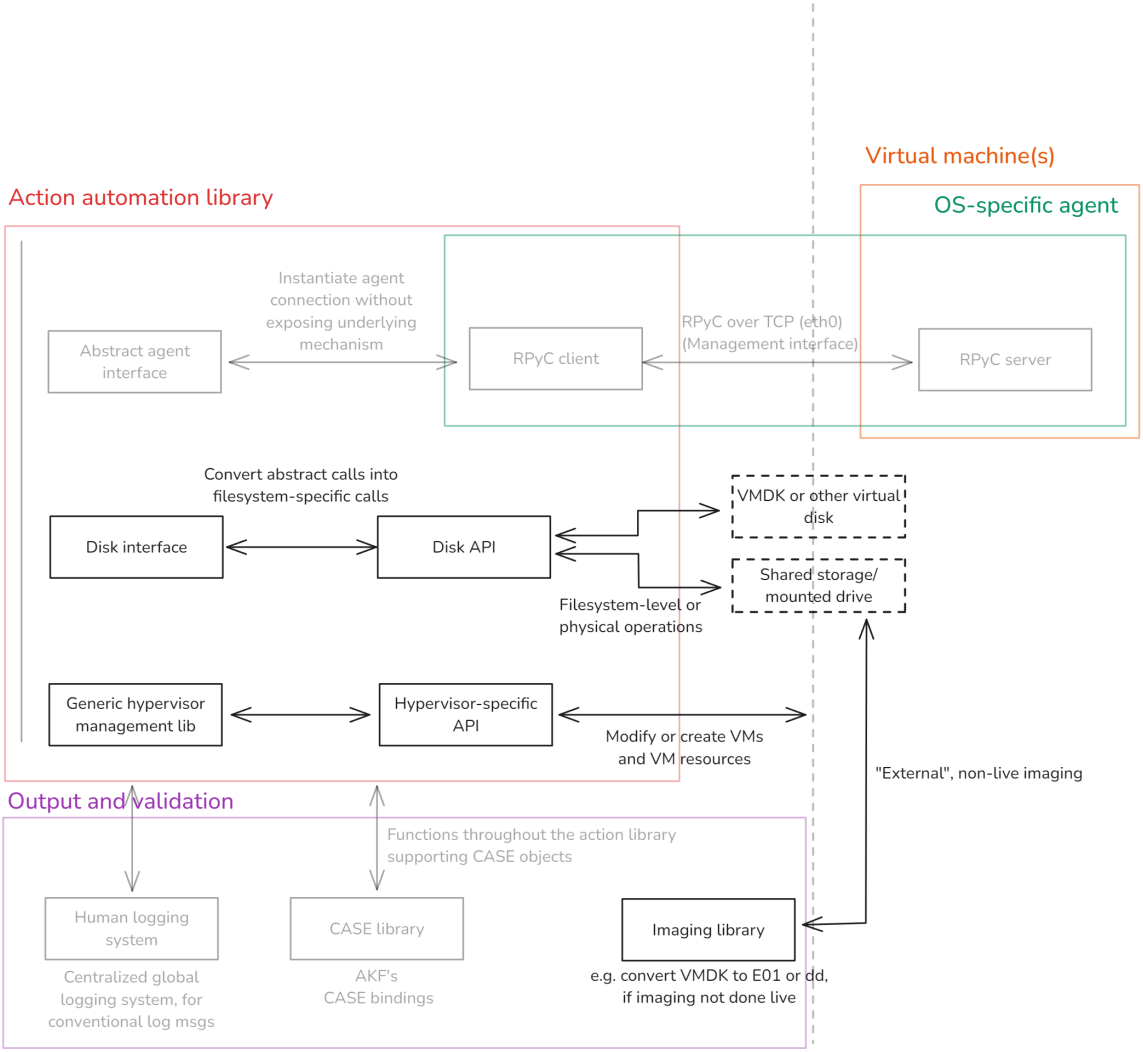

Figure 4.4: Abridged submodule diagram for agent-based artifact creation

First, AKF does not currently support mounting and writing to arbitrary filesystems or disk image formats, though this could be implemented using external image mounting software with the assistance of filesystem-independent libraries such as PyFilesystem [75]. However, AKF allows users to construct new ISO files from an existing host directory using the `pycdlib` library [76]. These ISOs can be used as stan-

dalone artifacts (for example, presenting them to students as disk images of removable drives) or as removable storage on a running virtual machine. Mounting these ISOs, as well as setting up a shared network folder between the host and guest machines, are the simplest means through which scenario developers can transfer files onto a running virtual machine.

Second, AKF also trivially "supports" filesystem-independent direct editing. Python allows users to open files in binary mode, at which point the user can call `seek()` on the file pointer to advance the pointer to a specific offset in the file. The user can then call `write()` to overwrite an arbitrary number of bytes at that position, achieving the same outcome as many other synthesizers that implement range and offset-based disk image editing.

Although writing to arbitrary positions is not always viable for correctly placing artifacts on a disk image, it is still a valid approach when the underlying filesystem has already been analyzed and the exact offsets and ranges to write are known. This may be the approach used by EviPlant [47] to construct and leverage its evidence packages for students; because filesystem analysis is costly and requires additional libraries not needed for offset-based editing, it is easier to perform this analysis at construction time and distribute the offsets to which data must be written to base images.

However, the filesystem may not have been analyzed ahead of time such that the exact offsets and data needed to place a file on disk are known. This brings us to filesystem-aware direct editing, in which AKF makes significant improvements.

ForGe implements its physical artifact generation by implementing NTFS and FAT32 data structures in Python, allowing it to create a fully virtual representation of these filesystems (with the assistance of a custom C program). This virtualized filesystem provides ForGe with the information necessary to efficiently insert data

into known slack and unallocated space while maintaining filesystem consistency. While extremely powerful (and implements a valuable feature not found in any other synthesizer to date), it is inflexible in two specific aspects:

- ForGe does not provide a generic interface for the filesystems it supports. Although the NTFS and FAT32 wrappers provide the same methods with the same signatures, this is not strictly enforced by a parent class. The lack of a generic interface means that the functionality supported across all filesystems is unclear, as is the functionality that must be implemented for new filesystems to be compatible with ForGe.
- ForGe lacks a "frontend" to support arbitrary disk types, regardless of the underlying filesystem. ForGe does not support multi-partition disks or common non-raw disk formats such as VHD, VMDK, or VDI.

Read-only libraries addressing these two issues have been in development since the introduction of ForGe but have not been integrated into other synthesizers to achieve the same write capabilities as ForGe. One such library is `libtsk` (also known as The Sleuth Kit), the C++ library that powers the open-source digital forensics platform Autopsy [77]. `libtsk` allows users to navigate and analyze the low-level contents of a variety of filesystems, including filesystem-specific attributes and the sequence of disk clusters that form a file. `libtsk` supports a large variety of filesystems through a filesystem-agnostic interface, including the FAT family of filesystems, the ext family, NTFS, and a variety of other filesystems. It also supports various volume systems for multi-partition disks, such as GPT and MBR, as well as several filesystem containers and images.

Python bindings for `libtsk` have existed for at least 15 years, with the most commonly used library being the automatically generated `pytsk` [78]. However, `pytsk` has relatively limited Python documentation and adoption, in addition to inheriting var-

ious known issues and limitations in `libtsk`, such as support for niche filesystems and partitioning formats. Many of these gaps were gradually covered as part of the `libyal` project, a collection of DFIR libraries developed by various contributors - most notably Joachim Metz from Google [79]. The `libyal` project includes individual libraries for analyzing formats not supported by `libtsk`, such as the QCOW disk image format used by QEMU, the Apple File System used by modern Apple devices, and more.

These libraries eventually led to the development of `dfvfs`, or the Digital Forensics File System, a Python library that leverages `libtsk` and multiple libraries from the `libyal` project to provide a generic interface for analyzing a variety of disk image formats and filesystems [80]. (The project is derived from Plaso and Google Rapid Response, two open-source DFIR tools used in various contexts.) Much like `libtsk`, it exposes a variety of low-level filesystem concepts common across multiple filesystems, such as the metadata and individual segments comprising a file at a known path on the filesystem. This low-level detail makes it extremely powerful in performing filesystem-aware direct editing, especially because of its focus on exposing this detail through a consistent Python interface.

AKF uses `dfvfs` to locate the clusters of a file at a known path in a filesystem, which can then be used to identify the start of slack space within the file's clusters (as well as unallocated space in the filesystem). By adding the physical offset of the cluster within the filesystem to the offset of the filesystem's partition in the disk image, AKF can write to the exact location of a file's slack space using the offset-based method described earlier. This achieves feature parity with ForGe, completely delegating filesystem and image-specific details to `dfvfs` and allowing for a filesystem-independent method for locating slack space.

More generally, this technique provides a deterministic method for inserting data within the slack space of a filesystem, simulating the deallocation of a file and its

partial replacement with a known file. However, this does not fully simulate the process of deleting a file through a running operating system and having a new file replace the deallocated clusters; naturally, this does not generate OS-specific artifacts associated with deleting and creating files and fails to generate the filesystem artifacts that could exist with the original file (such as the "deleted" file's name in the NTFS master file table). Future work in the field could address this gap by combining `dfvfs` with additional technologies to improve the accuracy of physical techniques.

It should be noted that this technique does not work on filesystems with full disk encryption enabled, as is the default on Windows systems beginning with the Windows 11 24H2 update. Additionally, this technique has only been verified to work on raw disk image formats. From limited testing, the offsets of partitions and files on compacted formats such as VHD are reported by `dfvfs` as if the partition were not compact, which makes it difficult to correctly determine the physical offset of a file on a VHD image. Additional research is needed to develop a reliable method for performing filesystem-aware physical artifact generation on such disk image formats.

Chapter 5

Output and validation

This chapter addresses the mechanisms through which AKF generates and documents its outputs, as seen in Figure 5.1. More specifically, this covers the modules and submodules seen in Figure 5.2. The submodules for the *virtual machine* and *action automation library* have been omitted for brevity.

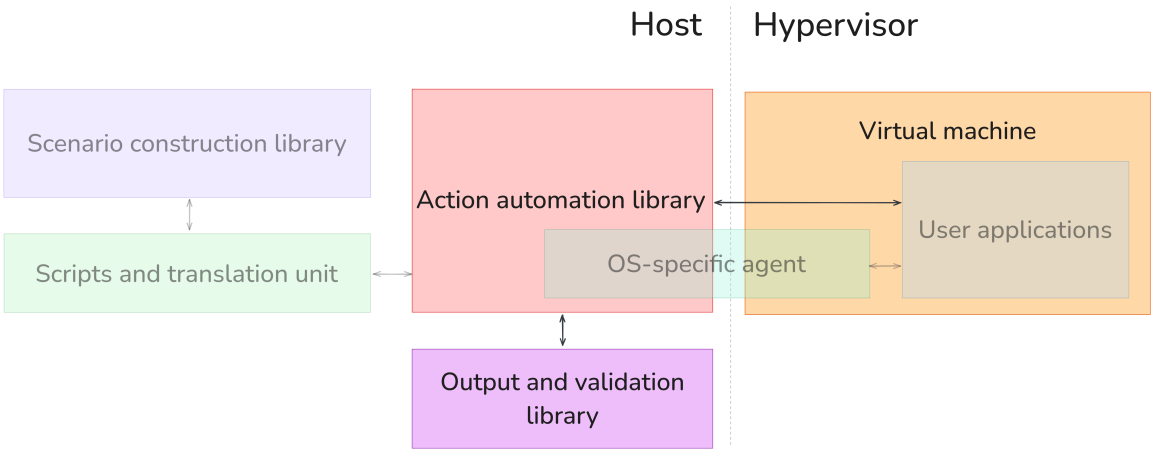

Figure 5.1: Simplified AKF architecture diagram for output and validation modules

Whether generating artifacts through physical or logical means, these artifacts

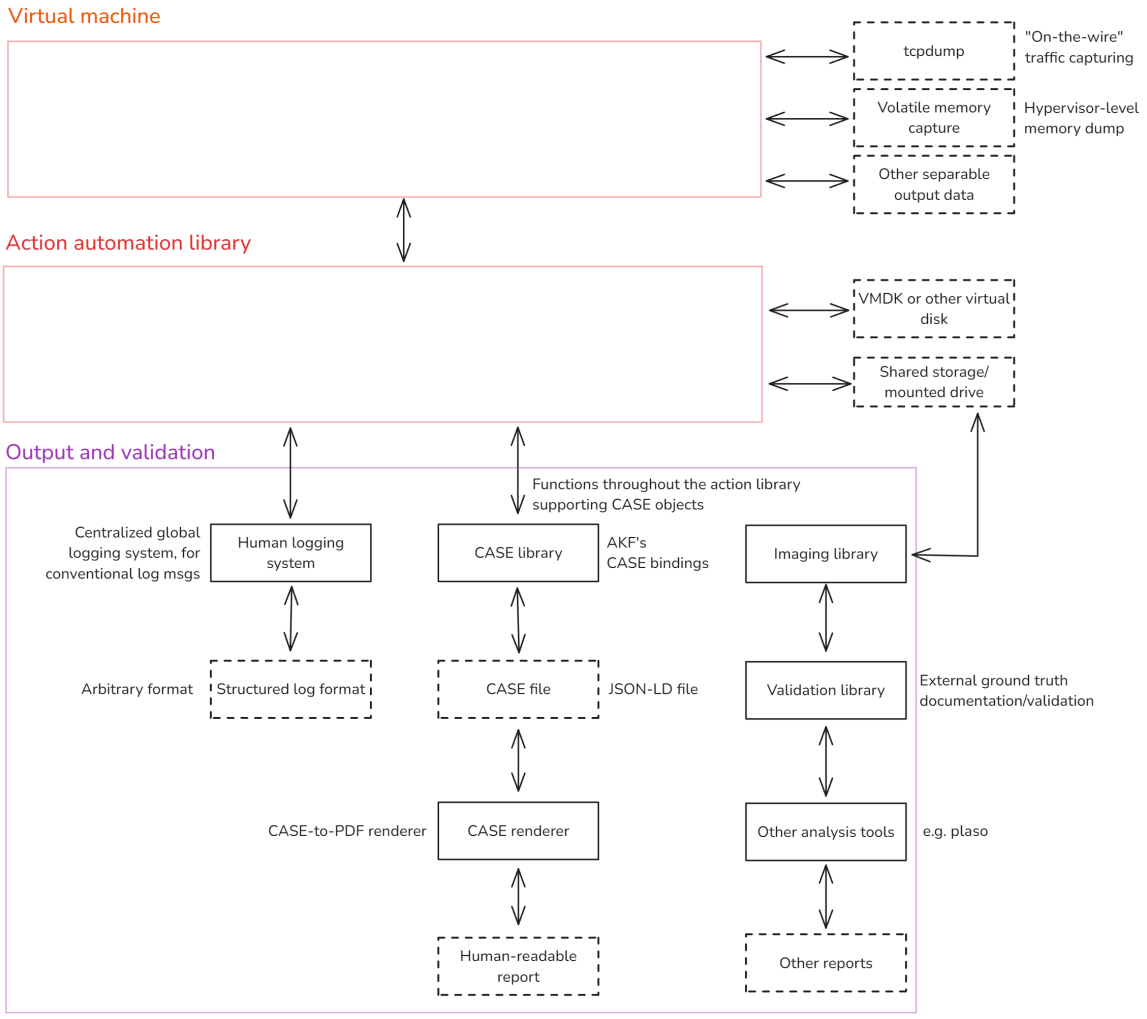

Figure 5.2: Abridged diagram of AKF modules related to output and validation

must ultimately be exported and documented. In most cases, this involves generating disk images, volatile memory captures, or network captures; additionally, individual artifacts, such as browser history, may be selectively extracted from a filesystem. Disk, memory, and network captures will be referred to as "core outputs" throughout this chapter to distinguish them from "standalone" artifacts. Both can be broadly described as "outputs" of a scenario.

The contents and details of these artifacts, as well as the means through which they were generated, should be documented in a structured format. This ground

truth should fully describe the contents of a forensic dataset; that is, it should identify every artifact that can be discovered within a dataset and every action taken to plant those artifacts. This metadata, in addition to core outputs and standalone artifacts, comprises a forensic dataset.

From an educational perspective, the ground truth represents an "answer key" to the dataset; it details every artifact of interest that an analyst could be expected to discover. For research, it allows for well-labeled datasets that can be used for tool development, validation, and testing. Ideally, this should be generated independently of the input script used to construct the scenario. This allows *all* artifacts to be documented, including those not explicitly declared or deemed significant by the scenario creator.

This chapter addresses the role of the output and validation library in providing several AKF services, including a centralized logging system, CASE object generation, and the generation of outputs such as disk images, network captures, and volatile memory dumps. In particular, it describes the reporting and documentation functionality enabled by CASE and the role that this metadata plays in ensuring dataset reproducibility and supporting community usage.

5.1 Core outputs

In the same way that artifacts can be generated through logical and physical means, outputs can also be generated through logical and physical means. Some of these are analogous to techniques used by real-world investigators to extract forensically sound evidence that is valid in a court of law; others are less suitable in formal settings but still have valid use cases.

Logical output generation refers to any technique in which software is used inside

a running virtual machine to generate these outputs. This includes using software such as FTK Imager [81] to capture the volatile memory of a device or construct a *logical* disk image. Similarly, network captures can be created by simply running Wireshark on the target device. Individual artifacts can be copied off the device by hand and sent to a network or removable drive.

Physical output generation refers to techniques in which the operating system is unaware of the technique, leaving few or no traces in the resulting outputs. In practice, this involves using tools such as hardware write blockers to extract disk images and network taps (or another traffic mirroring solution) to capture traffic over a particular interface. Although challenging, a physical extraction of RAM is also possible by performing a "cold boot attack." RAM sticks can be cooled to low temperatures before being removed from a running machine, slowing the process of memory decay as a result of the DRAM cells being unpowered [82].

AKF directly supports physical output generation for all three core outputs and indirectly supports logical output generation for core outputs and standalone artifacts. Physical output generation for virtual machines is generally achieved through direct interaction with the hypervisor.

- For **disk images**, the output can simply be the virtual hard drive used by the hypervisor on the host machine (such as VDI files for VirtualBox). If a standard raw disk image is desired, the command-line tool VBoxManage supports converting various virtual drive formats to raw disk images using the `vboxmanage clonemedium` command. This expands the compacted virtual drive to the "declared" size of the drive as seen by the operating system, allowing it to be bootable on actual hardware.
- For **network captures**, VirtualBox allows the user to enable network tracing over multiple interfaces, dumping network traffic as a .pcap file on the host

machine. This can be enabled or disabled at any time without affecting network connectivity or the state of the virtual machine.

- For **volatile memory dumps**, VBoxManage provides the command `vboxmanage debugvm <machine_name> dumpvmcore`, which creates an ELF core file of the virtual machine's memory that can be analyzed using a memory analysis tool such as Volatility [83].

These physical output options are often sufficient to generate datasets. If only specific files are desired, the existing file transfer utilities provided by AKF can be used to extract standalone artifacts. Users can also manually perform the logical techniques described above (such as running Wireshark or installing FTK Imager) through various means. For example, a user can pause an AKF script until instructed to continue; during this time, the user can manually run FTK Imager to extract the contents of RAM.

The logic for generating physical outputs is contained in the opaque submodules seen in Figure 5.3.

5.2 Metadata and ground truth

There exists a gap in the ability of instructors and researchers to perform bulk searches for specific forensic artifacts in public datasets. For example, the NIST CFReDS repository [30], one of the largest listings of forensic datasets, does not have a unified standard for describing uploaded images. Although users can search by keywords and human-applied tags, metadata is not available in a standardized format.

For many datasets, an instructor or researcher must read through a PDF answer key (if one exists) or analyze the image themselves to determine if a particular artifact is present. Answer keys are not inherently machine-readable and are not suited for

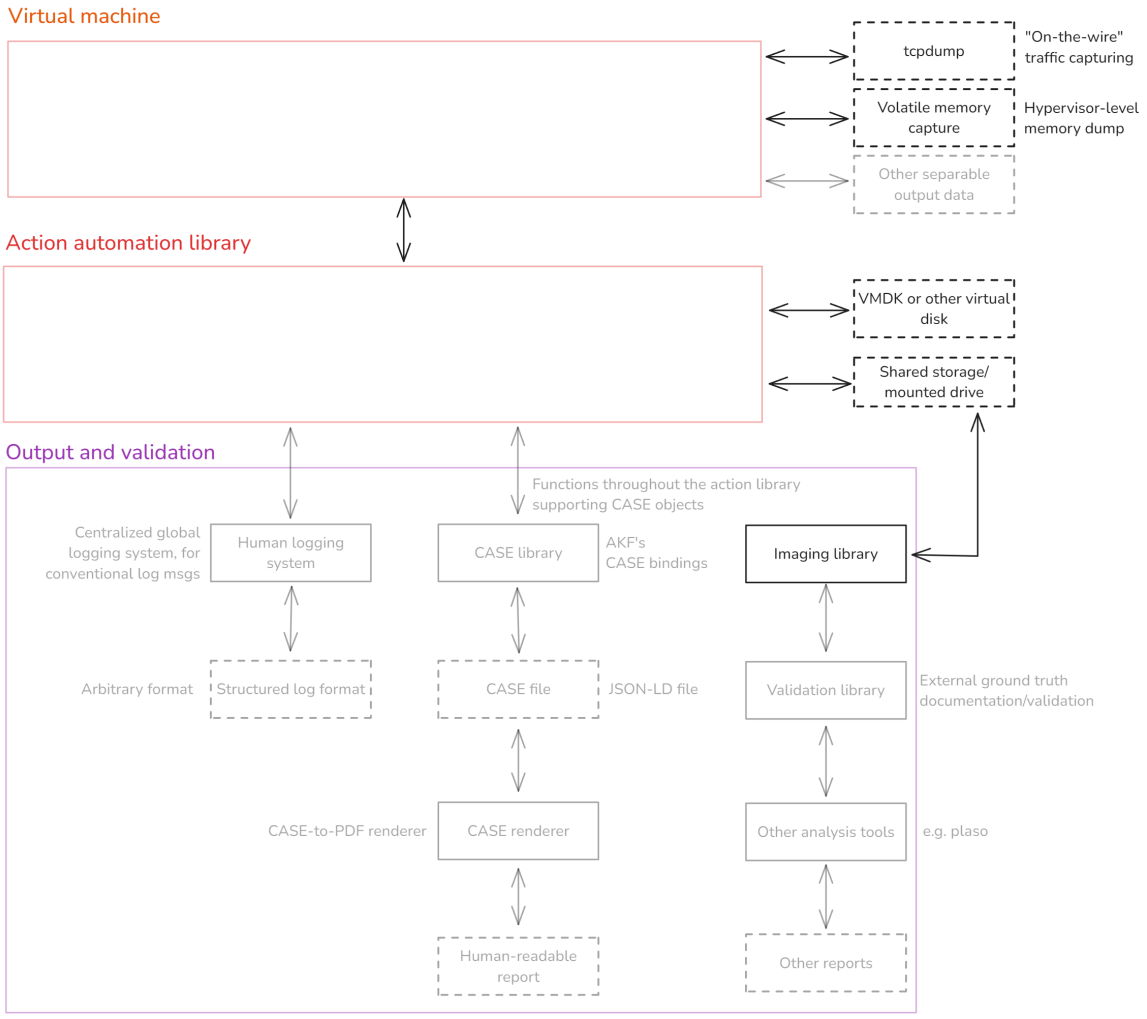

Figure 5.3: Diagram of AKF modules related to physical output generation

identifying specific artifact types in bulk. Additionally, the content of human-made reports may be limited to what the author believes is significant, even if other artifacts of interest are present in the image. In turn, it may be difficult to quickly determine if a dataset is useful in demonstrating a particular technique to students or validating a specific feature of a newly developed tool.

A rigid, well-defined format for ground truth is invaluable to researchers engaging in tool validation and development. Perhaps the lack of labeled forensic datasets on major repositories can be attributed to the lack of a need for one; Grajeda et

al. demonstrated that scenarios are rarely shared between researchers, so there is seldom a need to label them for general-purpose usage. AKF has the opportunity to solve this issue by allowing for the mass production of labeled datasets adopting a single established standard.

Many forensic analysis tools support exporting investigation details in both proprietary and language-agnostic formats. For example, Cellebrite's UFED supports exporting to UFDR and XML files, Magnet Axiom supports exporting to XML, and Autopsy supports a variety of formats, including Excel, STIX, and HTML. Each format varies in structure and content and does not necessarily contain equivalent information for the same analyzed disk image using default settings. This is the primary challenge with using an existing format, particularly a proprietary format subject to vendor changes; certain artifact types may be missing or change without warning. Similarly, many prior synthesizers (including AKF) generate structured log messages during artifact generation, though this is rarely suitable for documenting the contents of a dataset.

There has been extensive work in other fields towards developing a structured ontology that describes relationships and low-level details. This includes the EVIDENCE project for criminal justice and the Structured Threat Information Expression (STIX) format for conveying cyber threat intelligence [84]. For example, STIX provides a standard set of objects that allows organizations to describe observed attacker techniques and associate them with specific pieces of malware, attack campaigns, or threat actors.

However, there have been few efforts to document the contents of a forensic scenario (disk images and related metadata) in a vendor-neutral manner. One such format was developed by Abbott et al., who provide an XML-based notation of events in a dataset [85]. However, the specification does not appear to be available or main-

tained as part of another project. Similarly, although not designed to document technical details in a dataset, Conlan et al. describe a taxonomy for describing anti-forensic techniques at a high level that could be integrated into a scenario description language [86]. Neither of these formats appear to be supported by community tools, nor do they fill the need for low-level, queryable dataset descriptions.

Besides limited community support for and adoption of existing formats, Casey et al. found that these formats lacked features such as parent-child relationships, user actions, and non-technical case information such as a chain of custody. In response, the same authors introduced the Digital Forensic Analysis eXpression, or DFAX, a language extending CybOX (the predecessor to STIX) for use in the digital forensics community. DFAX later became the Cyber-investigation Analysis Standard Expression (CASE) [48]. CASE is perhaps the most comprehensive and actively supported ontology available for digital forensics; contributors include NIST with support from the Linux Foundation. As a result, AKF uses CASE as its primary format for documenting datasets, deliberately including support for CASE throughout various libraries.

5.2.1 CASE and Python bindings

CASE is a vendor-neutral format designed to document both technical and non-technical information about a digital forensics case [48]. It aims to cover as many OS-specific and application-specific artifacts as possible while still providing the flexibility to describe artifacts from uncommon applications. In theory, data exported from any major vendor, such as Cellebrite, Magnet, or FTK, can be converted into a valid CASE file. CASE is an extension of the Unified Cyber Ontology, or UCO, which provides basic objects not specific to digital forensics (such as applications or users). Consistent with the CASE project's documentation, references to CASE and

CASE/UCO throughout this chapter refer to the same project.

CASE is built on the Resource Description Framework (RDF), a model for describing information using relationships. Two objects are linked using a predicate, with the set of three elements forming a "triple." This pattern allows for directed, labeled graphs to be expressed using RDF. The ontology of CASE objects is defined using the Terse RDF Triple Language, or Turtle, which allows these triples to be written in a simple text format. In many ways, the Turtle definitions can be seen as the class definitions for CASE objects; instances of these objects can be expressed in JSON-LD, an extension of JSON for linked data. A collection of CASE objects is known as a bundle; applications can add instantiated CASE objects to a bundle as needed.

Because the CASE format itself is language-agnostic, it is necessary to write language-specific libraries that allow for instantiating CASE objects. At the time of writing, the CASE project provides Python bindings for UCO/CASE version 1.4 [87]. Each unique object type is represented as a Python class, which can be instantiated to produce individual objects. However, this library has several limitations due to its design; for example, CASE objects are internally represented as a dictionary of strings rather than a set of instance variables. While this makes it easier to serialize CASE objects to JSON-LD dictionaries, modifying objects after they have been instantiated is extremely difficult.

It is also worth noting that each object in the CASE ontology appears to have been manually translated to its corresponding Python class definition. This is slow and time-consuming, especially given that a significant overhaul of UCO/CASE to version 2.0 is underway with new object definitions [88]; there appears to be no active effort to update the 1.4 bindings to 2.0.

AKF leverages and contributes Pydantic-based bindings for CASE. The founda-

tion for this system is the Pydantic library for Python, which allows developers to quickly define classes (referred to as "Pydantic models" or simply "models") with built-in schema validation and serialization based on Python type hints [89]. More broadly, Pydantic allows us to simplify the declaration of individual objects while providing runtime type validation and automatic casting.

Examples of CASE-related definitions and a detailed comparison of AKF's bindings compared to the existing CASE bindings can be found in section B.3. One notable example from this section is the simplification of a CASE object declaration from 35 lines in the existing CASE bindings to only three lines in AKF. These simple declarations are possible primarily because the conversion of objects to valid JSON-LD elements is deferred until serialization instead of converting objects to dictionaries immediately upon instantiation. This design choice allows us to centralize the serialization logic in a single parent class that all CASE objects inherit from. In exchange for slightly increasing the complexity of converting objects to a dictionary with the correct key names, we can massively simplify the logic for declaring individual CASE objects.

A script is provided with AKF's CASE bindings to automatically parse the RDF files and convert them to valid Pydantic models. It automatically converts XSD datatypes to their native Python types (or a custom wrapper type if a native type does not exist), correctly inherits classes, and automatically generates docstrings and Pydantic fields as applicable. The script also topologically sorts dependencies in the same file; an RDF object's "parent" class may be declared *after* its "child" class, which is disallowed in Python. This dramatically simplifies the process of maintaining Python bindings for UCO/CASE, as well as the overall design of the library for future needs.

Note that there are certain inconsistencies between the Turtle CASE definitions

and those of AKF. These issues are true of the public bindings for CASE/UCO 1.4 that are currently available, as well; this is not unique to AKF's Pydantic-based implementation. These are described in greater detail at the end of section B.3.

5.2.2 CASE integration in AKF

AKF's CASE Python bindings are integrated throughout artifact-generating libraries in AKF. This is depicted through the opaque submodules in Figure 5.4.

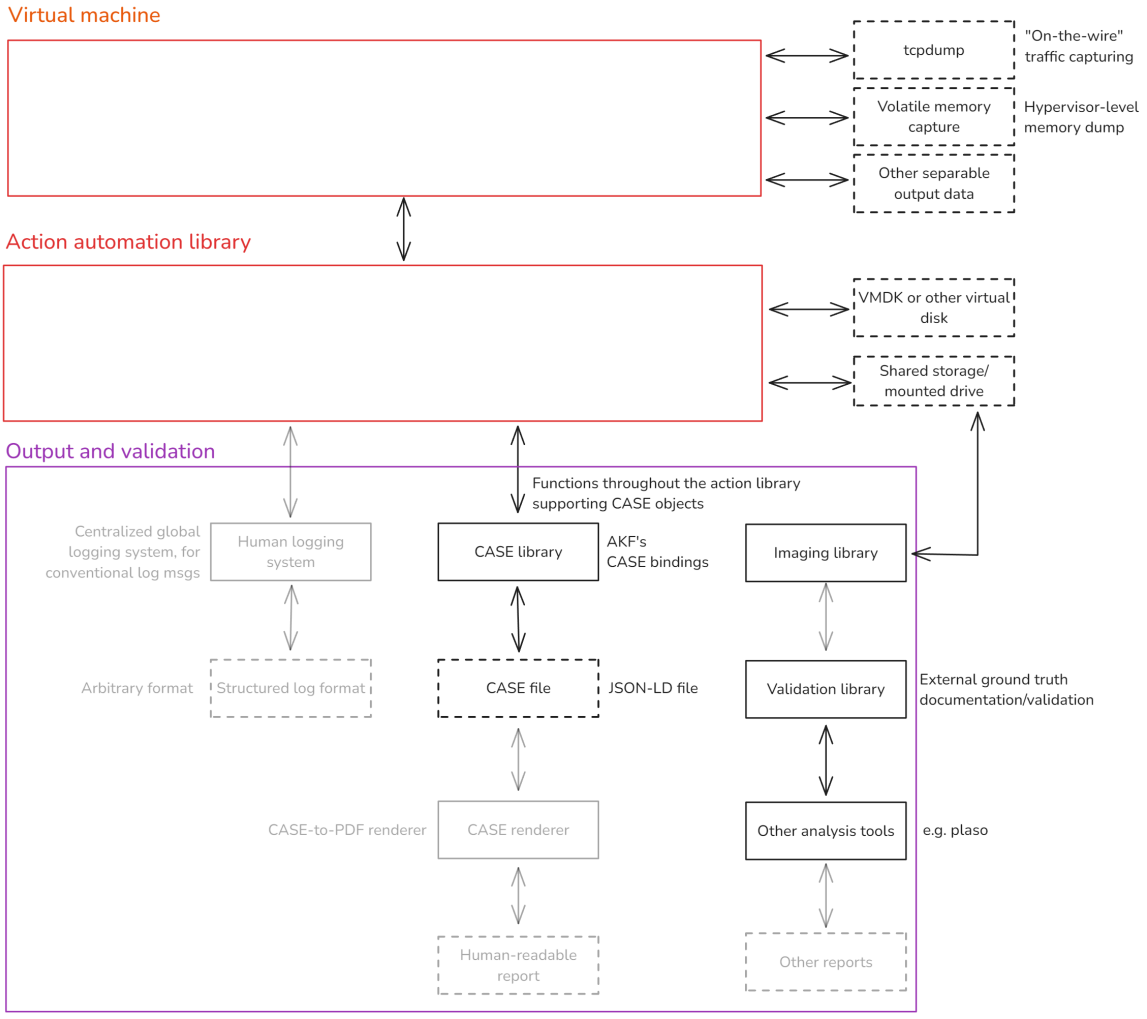

Figure 5.4: Diagram of AKF modules related to CASE object generation

There are two distinct points at which CASE artifacts can be generated and

attached to a scenario - during scenario generation and after scenario generation.

Various functions and classes throughout the AKF core libraries and agent API accept an optional CASE bundle when invoked or instantiated. As users call CASE-compatible functions, these functions can automatically add CASE objects corresponding to the artifacts generated through their execution. For example, if the agent subservice API for automating Chromium browser actions is instantiated with a CASE bundle, navigating to a page using the API will automatically generate a CASE object describing the page visit and add it to the bundle. This process can occur entirely within the host, allowing CASE-related logic to remain out of the agent where needed.

However, recall from subsection 4.2.2 that using RPyC for agent communication allows AKF to pass complex objects between the agent and the host machine. This includes CASE bundles and objects, as well. For example, suppose that a CASE object must be made for a file downloaded from the internet that frequently changes location, size, and content. The resulting CASE object can only be accurately constructed using some form of agent-side analysis as the action occurs. Once instantiated, the object can be returned to the host machine to append to a larger CASE bundle.

Some CASE ontology objects, such as those describing Windows prefetch files, are best constructed when disk images and other core outputs are created. For example, it is possible to create CASE prefetch objects during the execution of a scenario. However, these objects are likely to become outdated if their corresponding applications are launched later in the scenario, thus changing the content of the prefetch files and making the existing CASE objects inaccurate.

In turn, to make the generation of such objects more efficient and accurate, certain CASE objects must be instantiated "after the fact"; that is, they cannot or should not be automatically generated as part of AKF automation routines. This can be

trivially achieved by analyzing core outputs (such as disk images) using independent tooling. Such tools might include general-purpose DFIR tools that support CASE objects (such as Autopsy) or tools that explicitly focus on constructing CASE objects (such as tools built with the official or AKF Python bindings for CASE).

However, "after the fact" analysis can also be achieved on live virtual machines using existing AKF design patterns. In particular, it is possible to write RPyC subservices whose sole purpose is to generate CASE objects. For example, a subservice can collect the Windows prefetch files immediately after a disk image is created, allowing it to construct CASE prefetch objects that reflect the disk image without needing independent tooling. CASE-oriented behavior can also be implemented in existing subservices, such as adding a function to the Chromium subservice dedicated to identifying and creating CASE objects for all Chrome or Edge artifacts after all browser actions have been performed. This allows the dataset to be documented independently of synthesizer actions intended to emulate human activity – a desirable feature identified at the beginning of this chapter.

The flexibility of these two approaches – enabled by CASE's deep integration into AKF – makes it possible to construct CASE objects in a manner that requires little additional effort by scenario developers. Scenario developers do not need to be concerned with instantiating their own CASE objects when using high-level APIs so long as an AKF library developer has written support for automatic CASE object construction. This significantly reduces the need for scenario developers to construct artifact-specific ground truth through manual analysis of synthesizer-created outputs.

By extension, this means that the detailed documentation of AKF outputs is innate to many scenarios constructed using AKF. Lowering the effort required to document an AKF-generated scenario improves the likelihood that any public AKF scenario can be immediately valuable (or determined to be valuable) to researchers

and educators. This significantly contributes to AKF's goal of supporting an ecosystem around its images; the CASE bundles of many scenarios can be queried in bulk to identify datasets that might be useful for a specific purpose without having to download the dataset itself. This information can also be used to identify and analyze broader trends across scenarios, such as the frequency of a particular artifact appearing in all Windows datasets.

While this machine-readable reporting significantly improves the ability of the forensic community to locate useful datasets, it is verbose and unsuitable as a human-readable summary. Human-readable reporting is particularly relevant in a classroom setting, where the distribution of simplified answer keys to graders and students focusing on key artifacts is preferable to the exhaustive reporting provided by a CASE bundle. This leads us to the following section, which briefly addresses the conversion of AKF-generated metadata into human-readable reports.

5.3 Human readable reporting

As alluded to in the previous section, converting a rigid, well-defined format to a human-readable format is often easier than performing the reverse operation. Indeed, this is the approach taken by AKF, which does not create human-readable reports as an immediate output of dataset generation. (AKF generates human-readable log files during artifact generation, but these are unstructured and are created primarily for debugging rather than analysis.) Instead, AKF supports a simple yet flexible system for generating human-readable PDF reports from existing CASE bundles after generating a dataset.

AKF implements human-readable reporting through a set of "renderers," which focus on analyzing specific artifacts found in a CASE bundle and generating human-

readable content. Each renderer accepts a complete CASE bundle and extracts CASE objects of supported types; the renderer then uses the information contained in these objects to generate a Markdown document, which can include formatted text, tables, images, and other elements that may be useful to a human. The results of each renderer are combined to form a larger Markdown document (or documents) with multiple sections, one for each renderer. The combined document can be converted to a PDF using Pandoc [90], a general-purpose tool for converting between documents of various types.

A single CASE bundle can be passed through as many or as few renderers as needed to generate a suitable report for a dataset. So long as the original CASE bundle is available, users can reanalyze datasets with arbitrary renderers; this means that dataset reports can be regenerated with as much detail as a user needs for a specific use case. Furthermore, if new renderers are developed for artifacts that are present in older datasets, the human-readable report can be regenerated to include these artifacts.

This modular, "evergreen" approach to reporting allows these reports to be interpreted as a focused snapshot of what a dataset contains. Importantly, this can be done without compromising the dataset itself; the CASE bundle remains the single, comprehensive source of truth. Contrast this with human-written PDF reports, which may contain human biases and are rarely maintained in older datasets.

A sample report, generated from the AKF scenario described in section 6.2, is displayed in Figure 5.5:

Note that the displayed PDF uses the Eisvogel template [91] for Pandoc, significantly improving the appearance and readability of generated documents. Eisvogel is not distributed with AKF but can be manually installed alongside Pandoc.

After generating the scenario itself and any metadata and reporting that should

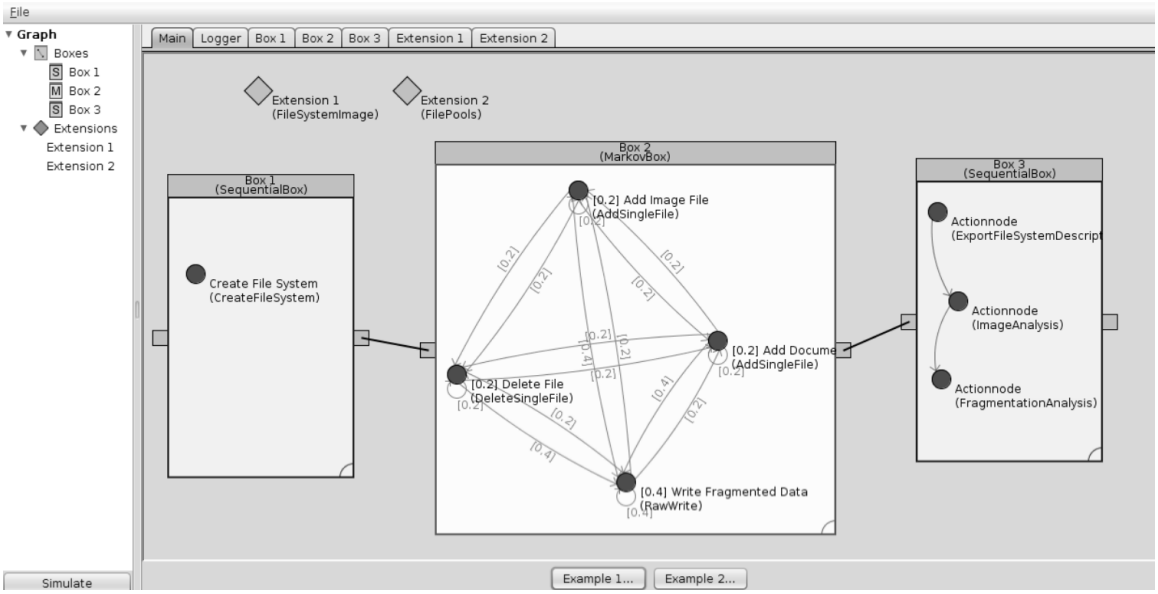

Figure 5.5: Sample PDF report generated by AKF renderers

be included with the dataset, the challenge of distributing this information remains. More precisely, how do we make our dataset as accessible, reusable, and discoverable as possible?

5.4 Distribution and community reproducibility

A key challenge identified by Grajeda et al. was the difficulty in reproducing results in the field of digital forensics. While this is primarily attributed to the *availability* of forensic datasets in general, it can also be attributed to challenges in the *reproducibility* of creating synthetic datasets.

Before addressing the low-level use of AKF as part of chapter 6, we briefly discuss the infrastructure needed to support community usage of the outputs of AKF scenarios and synthetic datasets as a whole. Note that for the remainder of this section, scenarios and datasets are both implied to be synthetic, as the principles of reproducibility are less applicable to real-world datasets.

There are four elements that must be distributed with a scenario to make a dataset (and its results) reproducible:

- Any core outputs or individual artifacts generated from the virtual machine.
- Any metadata, ground truth, or other reporting that describes the scenario.
- The OS-specific "base image" used to create the dataset, typically a virtual machine with a newly installed operating system on which all synthesizer actions are performed.
- The precise instructions required to build the scenario from the provided base image, whether human- or machine-readable instructions.

Forensic datasets have long included core outputs and individual artifacts well before the development of AKF and other synthesizers; there is limited value in a forensic scenario without anything to analyze. Various forms of ground truth have also long been a part of forensic datasets in multiple forms; some educational datasets include PDF answer keys, while some research datasets have been labeled in a structured format to include metadata about the dataset.

However, less common are detailed instructions to build the overall scenario. Manually constructed datasets rarely describe the actions taken to create a scenario in detail; for example, the educational M57-Patents scenario built by Woods et al. [44] provides an instructor PDF with a high-level timeline of actions taken in English. This detail is sufficient for educational purposes but is too imprecise to guarantee that others following this timeline will construct the image in the same manner as intended. As described in subsection 1.3.4, non-determinism can be acceptable and even desirable in educational contexts but is less desirable for tool validation and research.

Even rarer in manually constructed datasets is the inclusion of a base image representing the machine's state before any actions are performed. This may be

attributable to both copyright concerns and a perception that knowledge of the operating system alone is sufficient to rebuild the base image; while it is true that setting up a virtual machine is straightforward, any need for human interpretation introduces a source of non-determinism that could be eliminated.

Synthesizers significantly improve on the lack of precise instructions; their machine-readable scripts both document and execute the exact instructions needed to reconstruct a scenario. However, this depends on the availability of an OS- and synthesizer-specific base image; many synthesizers expect their users to follow a set of human-readable instructions to prepare a virtual machine specifically for use with that synthesizer.

Where copyright issues are not a concern, synthetic images should aim to include a complete definition of a virtual machine to be used as the base image. A base image may be a full, hypervisor-specific virtual machine (archiving and compressing the entirety of the associated virtual machine folder), a hypervisor-independent virtual appliance (in a format such as OVF), or another infrastructure-as-code solution to define and build virtual machines, such as Vagrant.

AKF is designed to provide all four of these elements in every scenario it creates; elements 1, 2, and 4 are inherent to all synthesizers, while base images can be provided as Vagrantfiles as described in section 6.2. This ensures that results from AKF-generated scenarios are reproducible from both a dataset creation and dataset usage perspective.

Although not explored as part of this thesis, the inclusion of all four of these elements as part of a well-structured, standardized distribution format could be used to build a distribution platform similar to CFReDS but with more powerful discovery and querying functionality. While the contents of the scenario are primarily described by CASE, it may also be possible to perform queries based on the contents of Va-

grantfiles and AKF scripts. For example, a user may want to search for all images that use the agent-based Chromium artifact generation described in subsection 4.3.2, which can be achieved by searching for the inclusion of the relevant AKF libraries in the scenario's scripts. However, this does not address the challenge of storing and distributing scenarios efficiently to support such a platform; this is discussed in section 8.4.

With the reproducibility and value of AKF-generated scenarios established, we now discuss how to invoke and leverage the underlying technologies that provide these benefits.

Chapter 6

Building scenarios

This chapter addresses the modules responsible for allowing users to invoke the framework through a standard Python script and a high-level YAML file. It also addresses the generative AI modules that can assist a user in building a scenario. These modules are depicted in Figure 6.1 below, with a more detailed diagram located in Appendix A as Figure A.5.

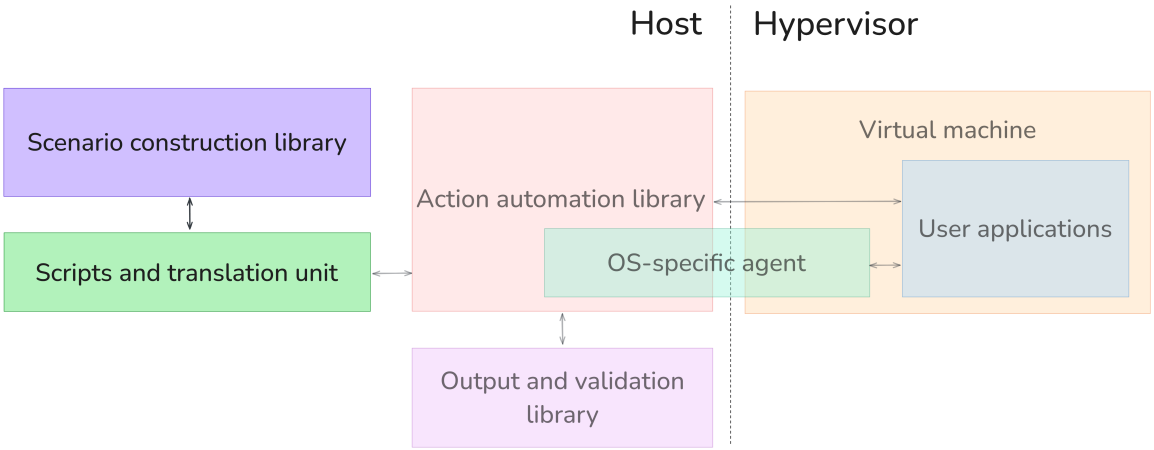

Figure 6.1: Simplified AKF architecture diagram for scenario construction modules

At this point, we have provided the implementations for automating artifact generation in a near-deterministic manner with comprehensive logging and reporting.

However, there is still the challenge of exposing this functionality in a user-friendly manner. At a high level, there are two primary ways to define an input to a synthesizer:

- An **imperative** format, in which the synthesizer is provided instructions in an imperative programming language, and the developer must provide the exact instructions for the synthesizer to take through some exposed API.
- A **declarative** format, in which the synthesizer is provided a file that describes the desired elements of the result, and it is up to the synthesizer to execute the instructions necessary to achieve the result.

These formats describe the *process* of placing artifacts and performing actions. However, the challenge of deciding *what* actions to perform remains. It is still mainly the responsibility of instructors to provide background noise and other realistic artifacts to insert into a scenario. Although the high-level languages provided by many of these frameworks make it easy to place files at desired locations or visit websites that are part of a scenario, these must all be defined and created ahead of time.

This chapter addresses the challenges of providing APIs for complex GUI-driven applications and creating background noise. More precisely, we address two questions – how do we invoke AKF's automation systems, and how does AKF assist a user in building a scenario? Here, we explore AKF's imperative and declarative syntaxes and the viability of using large language models (LLMs) to assist in building individual files and complete scenario descriptions.

6.1 Scripting background

We begin by analyzing how synthesizers accept instructions for execution – more precisely, how do users define the sequence of operations that the synthesizer should

take to create the dataset?

For many of the frameworks created in the last decade, users define scenarios by using a Python library to interact with the framework. The library is responsible for setting up the virtualized environment and performing high-level actions on the environment, abstracting away the underlying calls to the hypervisor from the scenario developer. This code-based approach represents an *imperative* strategy for scenario creation, where the user describes how the dataset should be created by defining the exact order and means to perform synthesizer actions. (It is worth noting that the language used to interact with the synthesizer's API does not need to match the language used to implement the synthesizer itself (although this is often the case). For example, the automation framework Playwright is implemented in TypeScript, initially exposing a Node.JS API [68]. Today, Playwright provides APIs in Python, Java, and C#.)

In contrast, custom scenario formats provided by D-FET [54], SFX [46], and Yannikos et al. [42] follow a different approach. For these synthesizers, a custom high-level language describes the desired final state of the dataset. Instead of importing libraries and writing code, users state the desired elements of the forensic dataset, allowing the synthesizer to decide how to create the desired dataset – a *declarative* strategy for scenario creation. The specifics of state management and execution are delegated to the synthesizer.

Consider the following declarative SFX code taken from Russell et al. shown in Listing 6.1 [46]:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
<disk>
    <partition index="p1" hidden="0" size="48M" type="ntfs">
        <base os="windows7x64"/>
        <user username = "Gordon">
             <browserhistory browser="firefox">
```

```
                    <url link="bbc.co.uk" time="13:14:00 1 Jan 2013"/>
                </browserhistory>
            </user>
        </partition>
</disk>
```

Listing 6.1: Declarative SFX scenario with web browsing

Here, a Windows 7 partition is created as part of a larger disk image. The partition is loaded in a virtual machine to create a user called "Gordon," who uses Firefox to browse the internet. (This simple web browsing scenario will be reused throughout this chapter to demonstrate several code examples.)

The same might be expressed in ForTrace [45] as shown in Listing 6.2, excluding additional code required for ground truth and disk image generation:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
import logging

from fortrace.core.vmm import Vmm
from fortrace.utility.logger_helper import create_logger
from fortrace.core.vmm import GuestListener

logger = logging.getLogger(__name__)

if __name__ == "__main__":
    logger = create_logger('fortraceManager', logging.DEBUG)
    macsInUse = []
    guests = []

    guestListener = GuestListener(guests, logger)
    virtual_machine_monitor1 = Vmm(macsInUse, guests, logger)
    # boottime expressed as "%Y-%m-%d %H:%M:%S"
    guest = virtual_machine_monitor1.create_guest(guest_name="w-guest01"
    , platform="windows", boottime="2013-01-01 13:14:00")

    browser_obj = guest.application("webBrowserFirefox", {'webBrowser':
    "firefox"})
    browser_obj.open(url="bbc.co.uk")
    while browser_obj.is_busy:
        time.sleep(2)
    browser_obj.close()
```

Listing 6.2: Imperative ForTrace scenario with web browsing [45]

Although these two code blocks have the same expressive power (that is, they

achieve the same overall outcomes), there is a clear difference in the complexity and length between them. It is significantly easier to read and write the declarative XML in the first code block, as it abstracts away the need to instantiate various synthesizer objects and call specific methods. By extension, this also allows for a common declarative syntax to be used across multiple synthesizers since low-level synthesizer details do not need to be exposed as part of the declarative syntax.

The primary benefit of an imperative approach to generation is its flexibility; on a Python-based synthesizer, one can simply import another library to extend the functionality of the base scenario definition. This flexibility naturally comes at the expense of a greater learning curve. Although many digital forensic specialists are likely to have programming experience, it is far easier to learn a restricted declarative specification (like XML) than an entire programming language, which may entail additional setup (such as installing an IDE, dependencies, and so on).

When accessibility is preferred over functionality, declarative syntaxes can be more valuable than imperative syntaxes. Some scenario developers, such as classroom instructors, may not need the low-level control provided by an imperative syntax or a complete programming language such as Python. It also takes time to learn about the functionality exposed by the synthesizer's library, not to mention learning the programming language itself. These are the primary motivators behind supporting well-defined declarative syntaxes.

Of course, low-level control is still important, especially when external libraries must be used to implement functionality not inherently exposed by a synthesizer. For this reason, some synthesizers support both declarative and imperative scripts to generate scenarios. For example, the Python-based hystck and ForTrace frameworks [45], [58] allow users to write YAML scripts to execute actions. Support for both formats can be implemented through various means, such as declarative-to-imperative

translators. (While not explored in this thesis, it is also worth noting the GUI-based interfaces provided by Yannikos et al. [42] and ForGe [56] for building scenarios.)

AKF supports an imperative syntax (through its Python API) and a custom declarative syntax. Unlike prior synthesizers, AKF's declarative syntax supports both execution and declarative-to-imperative translation, allowing users to quickly create and modify imperative scripts from high-level declarative descriptions.

6.2 Setup and basic usage

Like many of its predecessors, AKF implements its functionality and exposes its API in the same language, Python 3. There are numerous advantages to a Python-based API; besides the relatively low difficulty of setting up and using Python, its rich ecosystem allows scenarios to be extended through other libraries from the Python ecosystem. For example, if a user wanted to conditionally execute certain parts of a scenario by testing if a particular remote service is currently online, a user could use the Requests library [92] to issue an HTTP request out-of-band before performing the same action in a virtualized environment.

Users must install two foundational technologies for AKF to operate – Python 3.11 (or later) and a supported hypervisor. AKF currently only supports VirtualBox as its hypervisor, though QEMU/libvirt has also been used in prior synthesizers. AKF uses `pyproject.toml` to define Python library dependencies, which can be installed into a virtual environment using a package manager such as `pip` or `uv`.

At this point, a virtual machine must be prepared for use with AKF. As with prior synthesizers, it is possible to manually configure a machine by downloading a supported operating system and creating a new virtual machine from scratch. The manual process, which is similar to that of other synthesizers, is as follows:

- Download an ISO or pre-prepared virtual machine from a distributor with the desired operating system.
- If necessary, install the operating system on a new virtual machine.
- Configure the virtual machine with the desired host resources, including two network interfaces – one connected to the NAT adapter for general internet usage and one connected to the host-only adapter for agent communications.
- Create an administrative user with known credentials. Configure the operating system as desired to reduce friction with the synthesizer (such as disabling UAC prompts, enabling auto-logon, and so on.)
- Build and copy the OS-specific AKF agent to the virtual machine, configuring it as a startup application. Add firewall rules to ensure that the host and agent are able to communicate.

After this process, the virtual machine can be cloned and reused in multiple AKF scenarios. Although relatively straightforward, this process is still time-consuming, especially when adapting this process to new operating systems. While a prepared AKF virtual machine can theoretically be distributed (in a virtual appliance format such as OVF), this can run into legal issues if the software on the underlying operating system is copyrighted.

As a result, AKF uses modern infrastructure-as-code solutions to vastly simplify the setup of new virtual machines. Vagrant, developed by HashiCorp, is a tool for rapidly building development environments [93]. It allows users to define and build virtual machines on several virtualization platforms, including VirtualBox and VMWare. Virtual machines are built by configuring a base image according to a Vagrantfile, which describes hypervisor-specific configuration options and instructions to configure the machine. The Vagrantfile can be distributed to users, allowing them to build the same virtual machine without distributing virtual drives. (A similar

approach of distributing the "differences" of base images is used to reduce the size of distributed forensic datasets by EviPlant [47], as described in section 8.4.)

The AKF Windows agent includes a Vagrantfile for creating a new Windows 11 virtual machine with the agent installed and configured, which can easily be adapted for other platforms and hypervisors. The Vagrantfile(s) used to generate a dataset should be included with the dataset itself to maximize reproducibility, as described in section 5.4. A robust ecosystem of Vagrant boxes for varying Linux distributions and Windows versions exists, many of which can be pulled from the Vagrant public registry [94]. When combined with the flexibility of Vagrant over multiple virtualization platforms, this can significantly improve the reproducibility and usability of AKF across many platforms. It should also be noted that Vagrant can configure and build larger environments with multiple machines. For organizations that can express corporate environments as Vagrantfiles, AKF could perform artifact generation at scale, allowing for incident response scenarios reflecting real-world networks and events.

Following setup, developers can build scenarios using the AKF core libraries (`akflib`) and the API of the platform-specific agent installed onto the virtual machine. This reflects typical imperative usage, in which environment setup, artifact generation, and output generation are handled explicitly through a script executed through the Python interpreter.

A simple AKF script achieving the same outcomes as the ForTrace and SFX scripts above follows:

6.3 Declarative usage

As described previously, declarative inputs are well-structured files with a fixed set of available actions – effectively forming an API – where each entry in the file specifies an action or artifact to be generated as part of a dataset. Individual entries may contain additional configuration data that modifies how that specific action or artifact is generated. An *interpreter* is responsible for parsing and acting on the entries in the declarative file.

Interpreters can act on declarative formats in one of two ways. The first is *execution*, in which the elements of the declarative script are directly interpreted to generate imperative API calls. This is characteristic of synthesizers that only support declarative script inputs, exposing no low-level APIs. Execution takes advantage of the high-level nature of declarative scripts; a declarative script can remain the same even if the libraries that execute it change, so long as the interpreter is updated accordingly. The second is *translation*, in which the declarative script is used to generate an equivalent imperative script adhering to a particular synthesizer's API. This allows the declarative script to be used as a "base" for creating imperative scripts, such that an experienced scenario developer can modify the generated imperative script as needed. Imperative scripts can be regenerated from the same declarative script to reflect updates in library usage so long as the interpreter is updated accordingly, inheriting the perennial nature of declarative scripts.

It is important to note that a declarative syntax, which may be more "rigid" in structure, does not preclude the use of non-determinism or randomness. One notable example of this is the discrete-time Markov chains used by Yannikos et al. to express scenarios in a probabilistic manner, with each state of the Markov chain representing a particular action taken by the synthesizer (such as sending an email or deleting a file) [42]. These chains are evaluated at runtime to generate multiple unique datasets

from a single description.

The challenges of defining a suitable declarative syntax for a particular synthesizer are not unlike the challenges faced in general programming language design. There are several key factors to the success of imperative programming languages that extend to declarative syntaxes, some of which are derived from Finkel and described as follows [95]:

- The language should be **simple**, using as few basic concepts as possible. This makes code easier to read and write, an important aspect for users with limited programming experience.
- The language should be **modular**, such that the role and interfaces of individual program units are clear.
- The language should be **predictable**, such that users can apply their existing knowledge of a synthesizer to quickly implement or add new features to a scenario.
- The language should **abstract** as much as possible away from the user, such that the minimum information needed to fulfill artifact generation is exposed to the user.

The most important factor, however, is an awareness of the *purpose* of the declarative syntax. The purpose of a synthesizer is to make it easier to generate forensic artifacts and datasets. The declarative language should reflect this, focusing on making actions and artifacts as easy to declare and customize as possible.

In designing the AKF declarative syntax, the declarative syntaxes of prior synthesizers and unrelated technologies were evaluated. An analysis of some of these syntaxes, with examples, is described briefly in section B.4. However, two syntaxes in particular contributed the most to the AKF declarative syntax: those of ForTrace and Ansible, described in the following section.

6.3.1 Existing declarative syntaxes

ForTrace [45] uses YAML to express scenarios in a declarative manner. ForTrace significantly influenced both the AKF declarative syntax itself and the implementation of the declarative interpreter, primarily because it was the sole open-source synthesizer with both imperative and declarative support. Listing 6.3 demonstrates a simple example of a ForTrace declarative scenario:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
name: haystack-example
description: A example action suite to generate a haystack (traffic)
author: MPSE Group
seed: 1234
collections:
  c-http-0:
    type: http
    urls: ./generator/friendly_urls.txt
settings:
  host_nfs_path:
  guest_nfs_path:
applications:
hay:
  h-http-0:
    application: http
    amount: 3
    collection: c-http-0
needles:
  n-http-0:
    application: http
    file: [https://dasec.h-da.de/](https://dasec.h-da.de/)
    amount: 1
dumps:
  d-dump-0:
    dump-type: mem
    dump-path: /home/fortrace/gendump.file
```

Listing 6.3: Declarative ForTrace scenario with web browsing [45]

At a high level, ForTrace scenarios contain five distinct elements:

- Metadata about the scenario, such as the scenario's name, description, and author.
- "Collections" of data that can be reused throughout the scenario in supported

application types, such as a newline-separated list of URLs.

- Configuration options that may be applied to all actions of a particular action type or the entire scenario.
- The actual artifacts to create as part of the `hay` and `needles` sections, where `hay` includes artifacts that should be considered background noise, and `needles` includes artifacts that should be considered significant. Each artifact contains a unique ID, an application type (the `application` key), and arguments specific to the application responsible for generating the artifact, such as the URLs for web browsing.
- Any core outputs that should be created as part of the scenario.

This file is passed into a "generator," which parses the contents of the YAML file to prepare various internal data structures, initialize the virtual machine, and then execute the actions specified in the `hay` and `needles` sections in a random order according to the `seed` key. Depending on the value of the `application` key, the data for that action is passed to an application-specific handler that interacts with the running virtual machine using existing ForTrace libraries. Once all the actions have been executed, the generator creates any requested outputs (such as volatile memory dumps) and shuts down the virtual machine.

This analysis provided insight into the design decisions and functionality required to execute actions from the high-level descriptions of a scenario. In particular, it demonstrates the need for actions or artifacts to be defined consistently and flexibly so that program state and other data can be passed to application-specific libraries as needed. It also demonstrates the need for various levels of configuration, including scenario-wide configuration, application-specific configuration, and action/artifact-specific configuration. ForTrace implements these features in a somewhat awkward manner; in fact, nearly all declarative language support is contained in a single file,

with a hardcoded "router" handling each unique `application` type. This makes it difficult to add support for new applications without significant effort, particularly because the generator must be aware of every possible action/artifact type ahead of time.

With these priorities and issues from ForTrace identified, are there ideas from other technologies that can be used to address them? That is, are there other technologies designed to execute a large set of complex actions using a simple but flexible and configurable syntax, and how does it work? Ansible, the second major inspiration for the AKF declarative syntax, precisely fills this need.

Ansible [96] is an open-source automation framework often used to remotely configure Windows and Linux machines at scale, allowing organizations to manage many machines at once without installing orchestration software on these machines in advance. To achieve this, users write *playbooks*, which are simple YAML files that contain one or more *plays*. Each play is simply a set of *tasks* run on multiple machines simultaneously, and each task depends on a *module* designed to achieve a single, specific outcome.

An example of an Ansible playbook can be seen in Listing 6.4, which contains a single play to configure an Apache webserver.

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
1
2  - name: Update web servers
3    hosts: webservers
4    remote_user: root
5    tasks:
6    - name: Ensure apache is at the latest version
7      ansible.builtin.yum:
8        name: httpd
9        state: latest
10   - name: Write the apache config file
11     ansible.builtin.template:
12       src: /srv/httpd.j2
```

```
13      dest: /etc/httpd.conf
```

Listing 6.4: Minimal Ansible playbook for updating an Apache server [97]

This play uses the `yum` package manager to install Apache before copying a local configuration file to the remote host. `ansible.builtin.yum` and `ansible.builtin.template` are both modules, which accept parameters passed as a YAML dictionary. These modules are part of the `ansible.builtin` collection included with all default Ansible installations.

Although Ansible contains many features that contribute to its flexibility, the two important concepts relevant to AKF are *roles* and *modules*. Roles are a collection of Ansible resources that typically achieve some "larger" reproducible goal, typically by running multiple tasks and leveraging variables, configuration options, and files included as part of the role. Roles can include *modules*, which are standalone imperative scripts (typically in Python) that accept arguments, execute code based on those arguments, and return data using well-structured interfaces. As shown above, these modules can be called and executed from playbooks; they can also be executed independently on the command line.

These concepts are highly relevant to synthesizers, which must support application-specific actions and group these actions together in a flexible, well-defined manner. As described in subsection 4.2.2, each RPyC subservice of an AKF agent exposes a group of application-specific automation methods. The functionality of each of these groups must be re-exposed in a declarative manner, which can be achieved by adapting the concept of Ansible roles and modules to AKF.

Together, the ForTrace and Ansible syntaxes provide three concepts that are reflected in the AKF syntax, described further in the following section:

- The overall structure and contents of the declarative YAML file.
- An action syntax that allows us to declare individual actions, referring to those

actions by name, and pass arguments directly to that action for *translation* or *execution*.

- A modular architecture that allows us to define the supported arguments of each action and expose them to the declarative interpreter while also being decoupled from the standard imperative library as much as possible.

6.3.2 The AKF declarative syntax

The AKF declarative syntax is very similar to the Ansible playbook syntax. Declarative scripts are comprised of metadata, global configuration, libraries to import, and individual tasks to execute as part of the scenario. Each task refers to a single *module* by name using a qualified Python import path, accepting a dictionary of arguments in addition to global configuration overrides.

Like Ansible, individual modules are implemented as well-defined subclasses of `AKFModule`, an abstract base class that serves as the root of all AKF declarative modules. Each module must define Pydantic models that specify the arguments accepted by the module. The arguments declared in the YAML file are then passed to these module-specific Pydantic models for validation, after which the `AKFModule` must perform one of two tasks:

- **Execution**: When instructed to perform actions directly from the declarative script, the `AKFModule` should import AKF core libraries and agent APIs to perform the required actions.
- **Translation**: When instructed to translate the declarative script, the `AKFModule` should generate the equivalent code that *would* perform related actions if executed through a standard Python interpreter with the necessary libraries installed.

The ability of AKF to both execute and translate declarative scripts provides significant flexibility to scenario developers. To the best of our knowledge, prior synthesizers have only supported direct execution from declarative scripts, which limits the opportunities to use declarative scripts as a "starting point" for writing more complex imperative scripts.

In both cases, modules can read and modify a global state dictionary that allows otherwise independent modules to cooperate. For example, suppose that a module creates a new context manager in a `with` block, causing the indentation level of the translated code to increase. Successive modules can read the state dictionary to retrieve variable names and correctly indent generated code. Additionally, this allows for "outputs," such as CASE bundles, to be passed and gradually constructed across modules. This design allows for context-aware code generation and action execution.

An example of a declarative AKF scenario, carrying out the same actions as the SFX, ForTrace, and imperative AKF script above, can be seen in Listing 6.5:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
name: sample scenario
description: sample scenario
author: lgactna
seed: "0"
libraries:
  - akflib.actions
actions:
  - name: Run the sample action
    module: akflib.actions.sample.SampleModule
    args:
      arg1: "value1"
      arg2: "value2"
```

Listing 6.5: Example of a declarative AKF scenario

AKF declarative scripts, which are simply large YAML dictionaries, contain three distinct elements. The first is a set of high-level metadata keys associated with the scenario. The second is a set of scenario-wide configurations; in particular, it lists the libraries containing the necessary modules to execute or translate this imperative

script. Finally, scripts list a sequence of actions, typically a `module` specified by name and a dictionary of `args`.

The execution flow of the declarative interpreter itself is relatively straightforward. Given a path to a YAML script, the interpreter will load the necessary libraries and configuration keys defined in the file and instantiate resources accordingly. This may include setting global configuration variables, locating and starting a virtual machine by name, setting the `random` seed, and so on. Then, the interpreter runs each module under the `actions` key with the provided arguments and configuration in order, continuing until all `actions` have been completed.

Modules are located and executed using Python's dynamic import system. (For efficiency and safety, all modules in the script are located and "cached" at the start of script execution.) These modules can be located in any library so long as they can be found through Python's import system. For example, both `akflib` and the AKF Windows agent contain their own declarative module libraries, leveraging functionality specific to each code repository.

This design allows declarative modules to be written independently of the libraries they depend on, reducing the "impact" of supporting declarative features on the core imperative libraries. In fact, this independence allows for the AKF module system to be used in general-purpose scripting, similar to Ansible; it is not tightly bound to the creation of forensic scenarios and artifacts.

While the AKF imperative and declarative scripts provide users with significant flexibility in *using* AKF, the challenge of building artifacts and scenarios to generate through these execution options remains. The remainder of this chapter addresses this challenge.

6.4 Using generative AI for individual artifacts

Users of synthesizers must still perform a significant amount of work when generating individual artifacts. More precisely – while AKF and other synthesizers can streamline the process of placing artifacts on a dataset, users must still provide the actual artifacts themselves. For example:

- If a user wants to place 100 photos on the drive to simulate real usage, the user needs to create and provide 100 realistic images.
- If a user wants to simulate an email or other online conversation, the user needs to provide the entirety of the conversation to simulate.
- If a user wants to generate "proprietary" documents to emulate some form of corporate sabotage, the user would need to create and provide a variety of Microsoft Office, PDF, or other files in these formats.

This is particularly relevant when adding background noise often present in real-world datasets, namely the gigabytes of documents created as part of a user's benign activity over time. Although creating forensic datasets can be accomplished with the work presented thus far, creating realistic images that reflect real-world scenarios still requires extensive work. While it is true that images should often be small enough to allow students to explore a single specific technique, real-world scenarios encountered by analysts are rarely limited by time or size. An analyst might have to deal with a disk image used over a decade to store many photographs and send many messages. Such scenarios are valuable training material for courses that encapsulate a long period of forensic study, allowing students to apply many techniques learned throughout a forensic curriculum to analyze a realistic scenario.

With recent advancements in generative AI, popularized by services such as Midjourney and ChatGPT, it is now significantly easier to generate realistic images and

text content from short, high-level descriptions. Additionally, various services exist for creating realistic audio and video files that emulate a particular person's voice or facial movements; these can be used to generate additional scenario content of interest, especially if the scenario is based on a real-world event.

It holds that generative AI can quickly populate forensic datasets with realistic conversations and images consistent with an arbitrary scenario. For example, a corporate espionage case could be built by providing a large language model such as ChatGPT with prompts to describe complex machinery in both technical and conversational styles. Simultaneously, similar prompts can be passed into an image synthesizer such as Midjourney to produce related images. The images and text produced can then be used to create documents describing an unreleased product of high value, providing a pipeline through which significant artifacts can be placed onto a forensic dataset.

This idea can be extended further by training models on specific datasets; for example, if an instructor wished to create a fictional scenario in which a user frequently interacts with users of a particular online community, a large language model could be trained on available conversations to provide a degree of realism to the scenario. However, as mentioned before, this faces the challenges of ownership, privacy, and legality behind works derived from publicly available information that was (likely) not published with the expectation of its usage in an AI model.

It is important to note that the inclusion of generative AI into synthesizers does not necessarily require deep integration with the framework itself. Many existing synthesizers could be extended to use documents, images, or other data sourced from generative AI instead of user-defined files without the need to change the synthesizer's architecture. However, as advancements in AI continue, it may make sense to directly integrate AI-driven actions into synthesizers. For example, there may come a time in

which synthesizers can be provided natural language prompts (such as “Open Firefox and browse to news-related websites”) that directly lead to the generation of relevant artifacts without the need to explicitly program the process of browsing to a website in advance.

Chapter 7

Evaluation and observations

7.1 Context and scenario

7.2 Student analysis

Chapter 8

Future work

AKF was built with the expectation that it would be easy to maintain, develop, and extend. Indeed, there are several use cases that AKF does not fulfill as of writing, primarily due to time constraints. Future work related to AKF can be characterized as either tasks that integrate cutting-edge advancements to implement new features or tasks that extend or improve existing functionality. section 8.1 describes the application of recent AI developments towards addressing tasks relevant to forensic dataset development; the remaining sections focus on extending existing concepts in AKF.

8.1 Open-ended automation with AI

During the development of AKF, OpenAI announced the release of Operator, an "agent" capable of automating tasks on webpages using natural language prompts [98]. Rather than using a browser automation framework like Playwright or Selenium, it leverages its own browser to interact with webpages. Users can provide Operator with a high-level goal, which it then converts to concrete actions on its browser to achieve that goal.

OpenAI provides an example in which Operator searches Allrecipes for a clam linguine recipe and then orders the ingredients for the linguine through Instacart. Various sensitive actions during this process are delegated to the user to complete, such as inputting payment information or logging into a service. Operator is also capable of identifying ambiguity in a task and prompting the user for clarification, such as asking which store to use for ordering items through Instacart. This example demonstrates multiple notable features that are relevant to forensic synthesizers; in particular, it demonstrates the ability to both *interpret* and *interact* with arbitrary GUIs, as well as the ability to convert human prompts into a sequence of automated actions that may change as the agent discovers new information or encounters unexpected issues.

It is powered by what OpenAI calls its "Computer-Using Agent," or CUA, which is trained to interact with a virtual machine by accepting natural language and a screenshot of a virtual monitor [99]. It leverages OpenAI's GPT-4o model, following a three-step process in which it analyzes screenshots of the virtual desktop, conducts reasoning to determine the necessary steps to achieve a task (using prior context), and executes actions on the virtual machine. OpenAI notes that CUA can reliably perform simple tasks that a human would usually perform, such as navigating to specific categories of websites or repeating UI interactions. However, it struggles with some interfaces it has not encountered before and tends to be inefficient or hallucinate on more complex tasks.

The challenge of automating open-ended tasks through AI is not new. Multiple benchmarks (mentioned in OpenAI's articles), including WebVoyager, WebArena, and OSWorld, were developed in early 2024 to provide examples of typical webpage and OS interaction tasks performed by humans [100], [101], [102]. CUA is stated to achieve state-of-the-art results in these benchmarks, representing the current ability of

AI to address open-ended tasking. Although CUA has clear limitations and falls well behind human performance on these benchmarks, Operator demonstrates significant progress in the ability of AI models to replicate actions that are often performed by real users. Even in its current state, CUA can likely automate the "simple" tasking involved in generating background noise for forensic datasets, such as browsing news sites, interacting with social media platforms, and more.

In the near future, works similar to CUA or Operator will likely be capable of fully automating human actions as part of a larger scenario with high reliability and accuracy. Improvements to interacting with previously unseen GUI-based applications will increase the variety and complexity of actions that can be automated. These capabilities are extremely valuable in synthetic dataset generation, as they dramatically reduce the time needed to implement artifact generation functionality for novel technologies or developments.

What do these developments mean for "conventional" synthesizer architectures, which depend on scripts instead of natural language? First, the verbosity of writing a script and passing it to a synthesizer is still valuable, particularly in contexts where the non-determinism and opacity of an AI model may not be acceptable. Several works have described the non-deterministic nature of LLMs in multiple distinct tasks [103], [104], [105], which negatively impacts the reproducibility of results – an important quality of forensic datasets as described by Grajeda et al. [21]. Non-determinism is often acceptable in educational contexts so long as the actions taken by the model are logged and can be verified after the fact. However, the development of datasets for research and tool validation may require that a set of instructions always generates the same dataset every time, ensuring reproducibility from the instructions alone.

Second, these developments are not *incompatible* with synthesizers and should instead be seen as an option to complement them. The capabilities provided by

Operator could likely be built into AKF as part of its internal library or an OS-specific agent, providing users with access to both verbose imperative/declarative scripts as well as simpler natural language prompts when automating actions and building scenarios. Additionally, these agents cannot (currently) act as a substitute for physical artifact generation, in which the underlying filesystem or disk image must be edited to fulfill a particular task. Since these agents are trained to work with "live" operating systems and applications, they are likely unsuitable for automating physical operations on filesystems and disk images.

8.2 Alternative platform support

One future extension is implementing support for other desktop environments, such as Mac and Linux. At a high level, this requires implementing artifact generation methods described in chapter 4 for each platform.

Consistent with AKF's focus on using existing automation frameworks through agents to implement application-specific functionality, much of the effort for supporting other platforms requires writing and deploying a new OS-specific agent. Much of the code and overall design can be inherited from the Windows agent, enabled through the portability of Python and the lack of OS-specific assumptions made by RPyC. Certain automation frameworks may have significant cross-platform support (such as Playwright and `pywinauto`, which support Windows, Mac, and Linux), though implementing functionality for other applications may require more effort.

Implementing logical agentless generation through VirtualBox is likely straightforward, mainly because Oracle supports Guest Additions on MacOS and many Linux distributions. Similarly, the physical artifact generation implemented in AKF for FAT32 and NTFS (the most common filesystems for Windows) is likely extensible

to other disks, such as ext4 (the default for modern Linux distributions). This is because `dfvfs`, the library used to support disk image and filesystem editing, supports many filesystems, partitioning standards, and disk images commonly used by other operating systems.

The primary challenge in cross-platform support lies in accounting for artifacts or mechanisms unique to other operating systems. Some features, such as PowerShell/WinRM and Bash/SSH, are analogous between Windows and Unix-based systems and can be adapted accordingly. However, some concepts are truly unique to an operating system, such as performing modifications to the operating system through Windows registry keys, and may require more effort to adapt the same outcomes to other platforms.

8.3 Additional interface implementations

Another future extension involves developing new concrete implementations of AKF interfaces using other technologies that may have better support for specific host and guest platforms. AKF currently provides a VirtualBox implementation of the hypervisor-agnostic interface provided as part of `akflib`, largely depending on the VirtualBox Guest Additions software to function. Indeed, this is the approach taken by virtually all prior synthesizers that require virtualization; existing work leveraging VirtualBox to implement automation functionality has contributed significantly to its adoption across many synthesizers, including AKF.

However, prior synthesizers have also considered KVM/Qemu and VMWare as hypervisors; in particular, Forensig2 leveraged Qemu as its virtualization platform [38]. Several considerations exist for using other synthesizers, including support on different host platforms, performance, and available functionality. For example, Hwang et

al. compared the performance and low-level features of multiple hypervisors, including Hyper-V, KVM/Qemu, and VSphere, finding significant variability between these hypervisors when running workloads and applications [106]. It may be the case that specific scenarios or host machines are better suited for other hypervisors or even a multi-hypervisor environment. It is worth noting that VMWare has guest-specific software similar to the VirtualBox Guest Additions; multiple Python libraries exist for interacting with guests on VSphere, such as `pyvmomi` [107].

Similarly, rather than create new implementations of existing interfaces, it may also be worth building new interfaces entirely. One example is the implementation of agents in languages other than Python. Using non-Python agents may cover needs for specific tasks (such as actions that can be automated using a library for which no Python equivalent exists) or specific deployment restrictions (such as avoiding specific forms of synthesis pollution). In general, the implementation for an agent can be written in any language, so long as a corresponding Python API exists; the burden would be on the author to handle any communication or discrepancies across disparate languages, such as writing a TCP-based protocol for issuing commands.

8.4 Distribution

One notable development of prior synthesizers not implemented as part of AKF is the various improvements to the distribution of generated datasets, particularly in environments with limited bandwidth or storage space. AKF primarily addresses challenges in the *creation* of forensic datasets rather than their *distribution*. This section addresses prior efforts to reduce the size of forensic datasets and how they could be integrated into AKF in the future.

The size of forensic datasets can be trivially reduced in various ways. For example,

a full forensic dataset can be compressed using a standard algorithm such as LZMA. Specific core outputs, such as disk images, can be stored in a dynamically expandable image format such as VDI or VHD. Such formats only use as much space as is currently used on the disk image itself, even if the size reported to the operating system is much larger. Such reductions are "lossless" in that they can be reversed to reflect what a forensic analyst would encounter using real hardware.

However, more aggressive "lossy" reductions can be performed to further reduce dataset size. These can be broadly described as one of two approaches – either irrelevant data is outright removed, or the end user must take additional effort to finish preparing the forensic scenario for use after downloading relevant files (which are significantly smaller than the finished forensic dataset itself). The first approach reduces realism in exchange for a smaller dataset size; the second approach requires additional processing time in exchange for reduced network transfer sizes.

The first method describes the process of simply removing data that is not deemed to be relevant for educational or research purposes. For example, suppose an instructor is interested in demonstrating only the recovery and analysis of SQLite databases in Chrome's AppData directory. For this scenario, there is no need to include files from most other directories on the filesystem. One option for achieving this is to use a logical disk image format, such as the AD1 format, which is a collection of files (much like a ZIP archive) instead of a collection of physical disk units (like a standard disk image).

A more notable approach to removing irrelevant information is described by Russell et al. as part of their work developing SFX [46]. Their approach, described as "partition squeezing," is based on the idea that the contents of files irrelevant to an educational scenario can be truncated to the size of a single filesystem cluster or block. (In some filesystems, a file can be further truncated to fit entirely within

filesystem metadata structures, such as resident files in NTFS.) This preserves the file and directory structure of the overall filesystem (thus preserving realism) while also removing data that is unlikely to be useful to students performing analysis. Thus, the educational value of the image remains the same, even if some data is missing. Russell et al. note that additional space can be saved by replacing irrelevant files with hard links to other files, particularly those deep within the filesystem that are highly unlikely to be analyzed in detail by students. This entirely eliminates the original contents of the file while preserving most of its attributes.

The use of logical images and partition squeezing, of course, is not suitable in all cases. In particular, logical images generally do not include slack or unallocated space. The partition squeezing approach destroys data that may be required for certain forensic analysis tools to function, as well as data that might be of interest to users of the dataset in other research or educational contexts than initially intended.

AKF does not provide any functionality for directly generating logical images or performing partition squeezing. However, generating logical images is relatively straightforward using a publicly available tool such as FTK Imager, so long as the scenario developer is aware of relevant directories that should be included – a process that might occur with AKF's reporting features in mind. Partition squeezing could be achieved by leveraging the filesystem-aware functionality of libraries such as `libtsk` and `dfvfs`, which is used as part of AKF's physical planting techniques described in subsection 4.3.2. However, a more straightforward approach is to mount the filesystem and truncate all irrelevant files to the size of a single block or cluster. Such functionality could be implemented either in AKF or as part of a wholly independent tool.

The second method describes various techniques in which end users must take additional actions after downloading relevant materials before a forensic dataset is

ready for use. One example described by Scanlon et al. is the use of "evidence packages," in which users are provided with a base image of a single operating system that can be reused to build multiple forensic datasets through EviPlant [47]. After a scenario developer has finished creating artifacts (whether manually or through a synthesizer), they can use EviPlant's "diffing" tool to determine and extract differences between the base image and the developer's current disk state. These differences can be distributed to users, who can then use EviPlant's "injection" tool on the previously distributed base image to recreate the final disk image.

This approach can described more generally as "differential imaging", in which only differences from some initial files are saved and transmitted to users. These differences are often significantly smaller than the disk images they are designed to build and are much easier to distribute than complete disk images. (This is very similar to the approach used by Vagrant to distribute and prepare reproducible virtual machines, as described in section 6.2.) In exchange, each user must spend additional processing time before the image can be used for analysis. EviPlant's injection tool operates through a combination of logical and physical planting, which requires time, software, and resources that are not required when using and distributing complete disk images. This is an acceptable cost in some cases; for example, not all students in a remote classroom may have high-speed internet, limiting the size of files that can be distributed. Again, although AKF does not implement differential imaging, this functionality can likely be implemented as part of an independent tool or module.

8.5 Mobile synthesis

Finally, although well outside the scope of this thesis, there is also the challenge of building a synthesizer for non-desktop platforms, particularly mobile devices. For

many people, mobile devices are the primary means of interacting with the digital world, causing them to play a unique role in investigations [108]. Mobile devices can contain data from many facets of one's life, including text messages, location history, images, application logs, and other information that can provide insight into an individual's actions during a period of interest [109]. This information can be combined with other investigative methods, such as desktop and conventional forensics, to form a better understanding of a larger scenario.

Naturally, this means that there is a need for mobile datasets in digital forensics, as well. Grajeda et al. identify a small number of existing mobile dataset collections, including those containing Android malware, Android application files, and smartphone disk images [21]. However, these datasets are far less common than their desktop counterparts. Compared to the hundreds of desktop disk images hosted on Digital Corpora and CFReDS, there are only around 25 mobile disk images on the same platforms. A mobile-specific survey conducted by Gonçalves et al. in 2022 found not only a low availability of mobile images but also a lack of background noise and recent applications that would be present in a modern investigation [110].

There has been limited work in developing forensic synthesizers for mobile platforms. Two notable examples are FADE [111], developed by Delgado et al. in 2022, and an unnamed framework developed by Demmel et al. in 2024 [59]. FADE operates primarily through physical artifact generation, which it achieves by extracting and mounting partitions from an emulated, rooted Android device using the Android Debug Bridge. It then modifies application-specific database files to create artifacts such as phone call entries and text messages.

In contrast, Demmel et al. generate data using a logical agentless approach, using a tool called AndroidViewClient (AVC) to send keypresses and touch gestures [59]. Unlike VMPOP, which uses hardcoded mouse movements to click on GUI-based

elements, this framework leverages AVC to expose all interactable UI elements and programmatically "touch" these elements once the desired element has been found. This allows it to implement more complex application-specific functionality, such as using WhatsApp and opening Google Chrome. There has also been broader work in automating actions as part of testing pipelines for Android applications [112], [113], [114], though these primarily address application-specific actions, rather than automating actions across the entire device. Of note is the lack of iOS-relevant synthesizer functionality, perhaps due to the greater difficulty of working on a closed-source operating system with fewer options for exposing internal functionality.

It is possible that *some* of AKF's architecture could be adapted to support mobile dataset generation, primarily by interacting with Android emulators and existing developer tools. However, the isolation around individual Android applications suggests that it may be difficult to use a logical agent-based approach to automating application activity; a Python agent running as an application is unlikely to have full filesystem access on a non-rooted device. For simple application-specific artifacts, using physical techniques and editing artifacts in an application's allocated directory may be more effective. Much like Windows, however, Android applications may generate both application-specific artifacts and system artifacts (such as system logs) located elsewhere in the filesystem.

More research is needed to determine viable options for constructing Android datasets. This is especially true for iOS, for which there are significantly fewer resources.

Chapter 9

Conclusion

Public forensic datasets are invaluable to advancing research and education throughout digital forensics. However, high-quality datasets are presently few in number and may not fit specific needs, motivating the development of new datasets. Constructing these datasets by hand is time-consuming and prone to errors, yet it continues to be the primary method through which new datasets are made. In turn, there is a need for synthesizers, which allow users to create datasets using high-level scripting languages that can automate many common actions. Although prior synthesizers have addressed the creation of specific forensic artifacts, there are still opportunities to improve their flexibility and usability while promoting their usage throughout the forensic community.

AKF introduces a modern approach to forensic synthesis through a modular architecture focusing on sustained development and community adoption. It provides significant advancements in artifact generation, logging and validation, and the overall construction of new scenarios. It improves artifact generation by integrating numerous technologies not leveraged by prior synthesizers, greatly simplifying the implementation of the overall architecture without compromising the breadth of features

available through the framework. By using a centralized logging architecture and the CASE ontology, AKF is able to generate detailed, queryable metadata of the datasets it generates, allowing users to quickly identify artifacts of interest in a dataset. AKF also exposes a simple declarative syntax for generating artifacts, allowing users to develop scenarios without writing code using the underlying Python libraries. Finally, it provides a demonstration of using LLMs to further streamline scenario development, both when constructing individual artifacts and when building a complete scenario.

Although there are still limitations to what AKF can accomplish, numerous opportunities exist to leverage existing and emerging technologies to extend the framework. These contributions have been made in the hope that they will advance research and education throughout digital forensics, allowing the community to fill known gaps in public datasets.

Bibliography

[1] M. Pollitt, "A History of Digital Forensics," in *Advances in Digital Forensics VI*, K.-P. Chow and S. Shenoi, Eds., vol. 337, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–15, ISBN: 978-3-642-15505-5 978-3-642-15506-2. DOI: 10.1007/978-3-642-15506-2_1. Accessed: Dec. 3, 2023. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15506-2_1.

[2] G. M. Jones and S. G. Winster, "An Insight into Digital Forensics: History, Frameworks, Types and Tools," in *Cyber Security and Digital Forensics*, M. M. Ghonge, S. Pramanik, R. Mangrulkar, and D.-N. Le, Eds., 1st ed., Wiley, Feb. 8, 2022, pp. 105–125, ISBN: 978-1-119-79563-6 978-1-119-79566-7. DOI: 10.1002/9781119795667.ch6. Accessed: Nov. 27, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/9781119795667.ch6.

[3] Robert Kominski, "Computer Use in the United States: 1984," U.S. Bureau of the Census, Washington, D.C., Current Population Reports 155, Mar. 1988. [Online]. Available: https://www.census.gov/history/pdf/computerusage1984.pdf.

[4] Robert Kominski, "Computer Use in the United States: 1989," U.S. Bureau of the Census, Washington, D.C., Current Population Reports 171, Feb. 1991. [Online]. Available: https://www.census.gov/history/pdf/computerusage1984.pdf.

[5] C. Hargreaves, "Digital Forensics Education: A New Source of Forensic Evidence," in *Forensic Science Education and Training*, A. Williams, J. P. Cassella, and P. D. Maskell, Eds., 1st ed., Wiley, May 18, 2017, pp. 73–85, ISBN: 978-1-118-68923-3 978-1-118-68919-6. DOI: 10.1002/9781118689196.ch6. Accessed: Feb. 13, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/9781118689196.ch6.

[6] R. Montasari, V. Carpenter, and R. Hill, "A road map for digital forensics research: A novel approach for establishing the design science research process in digital forensics," *International Journal of Electronic Security and Digital Forensics*, vol. 11, no. 2, p. 194, 2019, ISSN: 1751-911X, 1751-9128. DOI: 10.1504/IJESDF.2019.098784. Accessed: Dec. 2, 2023. [Online]. Available: http://www.inderscience.com/link.php?id=98784.

[7] P. Cooper, G. T. Finley, and P. Kaskenpalo, "Towards standards in digital forensics education," in *Proceedings of the 2010 ITiCSE Working Group Reports*, Ankara Turkey: ACM, Jun. 28, 2010, pp. 87–95, ISBN: 978-1-4503-0677-5. DOI: 10.1145/1971681.1971688. Accessed: Nov. 26, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/1971681.1971688.

[8] G. Palmer and The MITRE Corporation, "A Road Map for Digital Forensic Research," in *The Digital Forensic Research Conference*, Utica, NY, Nov. 6, 2001. Accessed: Dec. 2, 2023. [Online]. Available: https://dfrws.org/wp-content/uploads/2019/06/2001_USA_a_road_map_for_digital_forensic_research.pdf.

[9] M. Geiger, "Evaluating Commercial Counter-Forensic Tools," 2005.

[10] R. Harris, "Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem," *Digital Investigation*, vol. 3, pp. 44–49, Sep. 2006, ISSN: 17422876. DOI: 10.1016/j.diin.2006.06.005. Accessed: Nov. 27, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287606000673.

[11] P. Anderson et al., "A Comparative Study of Teaching Forensics at a University Degree Level.," Jan. 1, 2006, pp. 116–127.

[12] S. Srinivasan, "Digital forensics curriculum in security education," *Journal of Information Technology Education. Innovations in Practice*, vol. 12, p. 147, 2013.

[13] K. Nance, B. Hay, and M. Bishop, "Digital Forensics: Defining a Research Agenda," in *2009 42nd Hawaii International Conference on System Sciences*, Waikoloa, Hawaii, USA: IEEE, 2009, pp. 1–6, ISBN: 978-0-7695-3450-3. DOI: 10.1109/HICSS.2009.160. Accessed: Nov. 26, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/4755787/.

[14] K. Nance, H. Armstrong, and C. Armstrong, "Digital Forensics: Defining an Education Agenda," in *2010 43rd Hawaii International Conference on System Sciences*, Jan. 2010, pp. 1–10. DOI: 10.1109/HICSS.2010.151. Accessed: Nov. 27, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/5428493.

[15] G. A. Dafoulas and D. Neilson, "An overview of Digital Forensics Education," in *2019 2nd International Conference on New Trends in Computing Sciences (ICTCS)*, Amman, Jordan: IEEE, Oct. 2019, pp. 1–7, ISBN: 978-1-7281-2882-5. DOI: 10.1109/ICTCS.2019.8923101. Accessed: Mar. 30, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/8923101/.

[16] L. Luciano, I. Baggili, M. Topor, P. Casey, and F. Breitinger, "Digital Forensics in the Next Five Years," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES '18, New York, NY, USA: Association for Computing Machinery, Aug. 27, 2018, pp. 1–14, ISBN: 978-1-4503-6448-5. DOI: 10.1145/3230833.3232813. Accessed: Mar. 30, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3230833.3232813.

[17] W. A. Conklin, G. White, C. Cothren, R. L. Davis, and D. Williams, "Computer Forensics," in *Principles of Computer Security: CompTIA Security+ and Beyond (Exam SY0-601)*, 6th Edition, New York: McGraw-Hill Education, 2022, ISBN: 978-1-260-47431-2. [Online]. Available: https://www.accessengineeringlibrary.com/content/book/9781260474312/chapter/chapter23.

[18] Bureau of Labor Statistics, U.S. Department of Labor. "Information Security Analysts," Occupational Outlook Handbook, Accessed: Dec. 3, 2023. [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/information-security-analysts.htm#tab-1.

[19] S. Garfinkel, "Forensic Corpora: A Challenge for Forensic Research," vol. 2007, Jan. 1, 2007.

[20] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digital Investigation*, vol. 6, S2–S11, Sep. 2009, ISSN: 17422876. DOI: 10.1016/j.diin.2009.06.016. Accessed: Nov. 26, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287609000346.

[21] C. Grajeda, F. Breitinger, and I. Baggili, "Availability of datasets for digital forensics – And what is missing," *Digital Investigation*, vol. 22, S94–S105, Aug. 2017, ISSN: 17422876. DOI: 10.1016/j.diin.2017.06.004. Accessed: Nov. 6, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287617301913.

[22] K. Withers J., "Electronically Stored Information: The December 2006 Amendments to the Federal Rules of Civil Procedure," *Northwestern Journal of Technology and Intellectual Property*, vol. 4, no. 2, p. 171, 2006. [Online]. Available: https://web.archive.org/web/20120321221012/http://www.law.northwestern.edu/journals/njtip/v4/n2/3/.

[23] G. M. Nist, "The NIST Cybersecurity Framework 2.0," National Institute of Standards and Technology, Gaithersburg, MD, NIST CSWP 29 ipd, 2023, NIST CSWP 29 ipd. DOI: 10.6028/NIST.CSWP.29.ipd. Accessed: Dec. 3, 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.29.ipd.pdf.

[24] S. Brueckner, D. Guaspari, F. Adelstein, and J. Weeks, "Automated computer forensics training in a virtualized environment," *Digital Investigation*, The Proceedings of the Eighth Annual DFRWS Conference, vol. 5, S105–S111, Sep. 1, 2008, ISSN: 1742-2876. DOI: 10.1016/j.diin.2008.05.009. Accessed: Aug. 31, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287608000406.

[25] K. R. Lawrence and H. Chi, "Framework for the design of web-based learning for digital forensics labs," in *Proceedings of the 47th Annual Southeast Regional Conference*, ser. ACM-SE 47, New York, NY, USA: Association for Computing Machinery, Mar. 19, 2009, pp. 1–4, ISBN: 978-1-60558-421-8. DOI: 10.1145/1566445.1566546. Accessed: Nov. 27, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/1566445.1566546.

[26] D. Chang, M. Ghosh, S. K. Sanadhya, M. Singh, and D. R. White, "FbHash: A New Similarity Hashing Scheme for Digital Forensics," *Digital Investigation*, vol. 29, S113–S123, Jul. 2019, ISSN: 17422876. DOI: 10.1016/j.diin.2019.04.006. Accessed: Dec. 4, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287619301550.

[27] G. Pessolano, H. O. Read, I. Sutherland, and K. Xynos, "Forensic Analysis of the Nintendo 3DS NAND," *Digital Investigation*, vol. 29, S61–S70, Jul. 2019, ISSN: 17422876. DOI: 10.1016/j.diin.2019.04.015. Accessed: Dec. 4, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287619301641.

[28] X. Lin, T. Chen, T. Zhu, K. Yang, and F. Wei, "Automated forensic analysis of mobile applications on Android devices," *Digital Investigation*, vol. 26, S59–S66, Jul. 1, 2018. DOI: 10.1016/j.diin.2018.04.012.

[29] National Institute of Standards and Technology. "Computer Forensics Tool Testing Program (CFTT)," NIST, Accessed: Dec. 4, 2023. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/computer-forensics-tool-testing-program-cftt.

[30] National Institute of Standards and Technology. "CFReDS Portal," Accessed: Dec. 4, 2023. [Online]. Available: https://cfreds.nist.gov/.

[31] F. Adelstein, Y. Gao, and G. Richard, "Automatically Creating Realistic Targets for Digital Forensics Investigation," presented at the Digital Forensic Research Workshop, 2005. Accessed: Nov. 28, 2023. [Online]. Available: https://www.semanticscholar.org/paper/Automatically-Creating-Realistic-Targets-for-Adelstein-Gao/750d289378fa28f7d78fa2959194fdf6756f55c6.

[32] K. Gupta, A. Neyaz, N. Shashidhar, and C. Varol, "Digital Forensics Lab Design: A framework," in *2022 10th International Symposium on Digital Forensics and Security (ISDFS)*, Jun. 2022, pp. 1–6. DOI: 10.1109/ISDFS55398.2022.9800799. Accessed: Nov. 27, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9800799.

[33] J. Park, "TREDE and VMPOP: Cultivating multi-purpose datasets for digital forensics – A Windows registry corpus as an example," *Digital Investigation*, vol. 26, pp. 3–18, Sep. 2018, ISSN: 17422876. DOI: 10.1016/j.diin.2018.04.025. Accessed: Sep. 17, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287617303614.

[34] G. Horsman and J. R. Lyle, "Dataset construction challenges for digital forensics," *Forensic Science International: Digital Investigation*, vol. 38, p. 301 264, Sep. 1, 2021, ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2021.301264. Accessed: Dec. 30, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721001815.

[35] I. Baggili and F. Breitinger, "Data sources for advancing cyber forensics: What the social world has to offer," ser. AAAI Spring Symposium - Technical Report, Mar. 2015.

[36] C. Meffert, D. Clark, I. Baggili, and F. Breitinger, "Forensic State Acquisition from Internet of Things (FSAIoT): A general framework and practical approach for IoT forensics through IoT device state acquisition," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17, New York, NY, USA: Association for Computing Machinery, Aug. 29, 2017, pp. 1–11, ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3104053. Accessed: Aug. 30, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3098954.3104053.

[37] R. U. Rahman and D. S. Tomar, "A new web forensic framework for bot crime investigation," *Forensic Science International: Digital Investigation*, vol. 33, p. 300 943, Jun. 1, 2020, ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2020.300943. Accessed: Aug. 31, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281720300718.

[38] C. Moch and F. C. Freiling, "The Forensic Image Generator Generator (Forensig2)," in *2009 Fifth International Conference on IT Security Incident Management and IT Forensics*, Stuttgart, Germany: IEEE, 2009, pp. 78–93, ISBN: 978-0-7695-3807-5. DOI: 10.1109/IMF.2009.8. Accessed: Sep. 17, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/5277859/.

[39] O. Avrahami and B. Tamir. "Ownership and Creativity in Generative Models." arXiv: 2112.01516 [cs], Accessed: Dec. 4, 2023. [Online]. Available: http://arxiv.org/abs/2112.01516, pre-published.

[40] J. K. Eshraghian, "Human ownership of artificial creativity," *Nature Machine Intelligence*, vol. 2, no. 3, pp. 157–160, 3 Mar. 2020, ISSN: 2522-5839. DOI: 10.1038/s42256-020-0161-x. Accessed: Dec. 4, 2023. [Online]. Available: https://www.nature.com/articles/s42256-020-0161-x.

[41] K. Roose, "An AI-generated picture won an art prize. Artists aren't happy," *The New York Times*, Sep. 2, 2022. [Online]. Available: https://cs.uwaterloo.ca/~jhoey/teaching/cogsci600/papers/Roose2022.pdf.

[42] Y. Yannikos, L. Graner, M. Steinebach, and C. Winter, "Data Corpora for Digital Forensics Education and Research," in *Advances in Digital Forensics X*, G. Peterson and S. Shenoi, Eds., ser. IFIP Advances in Information and Communication Technology, Berlin, Heidelberg: Springer, 2014, pp. 309–325, ISBN: 978-3-662-44952-3. DOI: 10.1007/978-3-662-44952-3_21.

[43] National Institute of Standards and Technology. "CFReDS - Data Leakage Case," CFReDS Archive, Accessed: Dec. 4, 2023. [Online]. Available: https://cfreds-archive.nist.gov/data_leakage_case/data-leakage-case.html.

[44] K. Woods, C. Lee, S. Garfinkel, D. Dittrich, A. Russell, and K. Kearton, "Creating Realistic Corpora for Security and Forensic Education," *Proceedings of the ADFSL Conference on Digital Forensics Security and Law*, Jan. 1, 2011.

[45] T. Göbel, S. Maltan, J. Türr, H. Baier, and F. Mann, "ForTrace - A holistic forensic data set synthesis framework," *Forensic Science International: Digital Investigation*, Selected Papers of the Ninth Annual DFRWS Europe Conference, vol. 40, p. 301 344, Apr. 1, 2022, ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2022.301344. Accessed: Aug. 31, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281722000130.

[46] D. G. Russell, R. Macfarlane, and R. Ludwiniak, "A Forensic Image Description Language for Generating Test Images," 2012.

[47] M. Scanlon, X. Du, and D. Lillis, "EviPlant: An efficient digital forensic challenge creation, manipulation and distribution solution," *Digital Investigation*, DFRWS 2017 Europe, vol. 20, S29–S36, Mar. 1, 2017, ISSN: 1742-2876. DOI: 10.1016/j.diin.2017.01.010. Accessed: Aug. 31, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287617300397.

[48] E. Casey, S. Barnum, R. Griffith, J. Snyder, H. van Beek, and A. Nelson, "Advancing coordinated cyber-investigations and tool interoperability using a community developed specification language," *Digital Investigation*, vol. 22, pp. 14–45, Sep. 1, 2017, ISSN: 1742-2876. DOI: 10.1016/j.diin.2017.08.002. Accessed: Dec. 3, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301007.

[49] K. Ricanek and T. Tesafaye, "MORPH: A longitudinal image database of normal adult age-progression," in *7th International Conference on Automatic Face and Gesture Recognition (FGR06)*, Apr. 2006, pp. 341–345. DOI: 10.1109/FGR.2006.78. Accessed: Mar. 5, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/1613043.

[50] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digital Investigation*, The Proceedings of the Ninth Annual DFRWS Conference, vol. 6, S2–S11, Sep. 1, 2009, ISSN: 1742-2876. DOI: 10.1016/j.diin.2009.06.016. Accessed: Feb. 8, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287609000346.

[51] W. Xu, L. Deng, and D. Xu, "Towards Designing Shared Digital Forensics Instructional Materials," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, Jun. 2022, pp. 117–122. DOI: 10.1109/COMPSAC54236.2022.00025. Accessed: Nov. 27, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9842462.

[52] G. Michelet, F. Breitinger, and G. Horsman, "Automation for digital forensics: Towards a definition for the community," *Forensic Science International*, vol. 349, p. 111 769, Aug. 1, 2023, ISSN: 0379-0738. DOI: 10.1016/j.forsciint.2023.111769. Accessed: Dec. 3, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0379073823002190.

[53] C. Moch and F. C. Freiling, "Evaluating the Forensic Image Generator Generator," in *Digital Forensics and Cyber Crime*, P. Gladyshev and M. K. Rogers, Eds., vol. 88, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 238–252, ISBN: 978-3-642-35514-1 978-3-642-35515-8. DOI: 10.1007/978-3-642-35515-8_20. Accessed: Nov. 26, 2023. [Online]. Available: http://link.springer.com/10.1007/978-3-642-35515-8_20.

[54] P. William et al., "Cloud-based digital forensics evaluation test (D-FET) platform.," Jan. 1, 2011.

[55] maxfragg, *Maxfragg/ForGeOSI*, Jun. 16, 2023. Accessed: Feb. 6, 2025. [Online]. Available: https://github.com/maxfragg/ForGeOSI.

[56] H. Visti, S. Tohill, and P. Douglas, "Automatic Creation of Computer Forensic Test Images," in *Computational Forensics*, U. Garain and F. Shafait, Eds., vol. 8915, Cham: Springer International Publishing, 2015, pp. 163–175, ISBN: 978-3-319-20124-5 978-3-319-20125-2. DOI: 10.1007/978-3-319-20125-2_14. Accessed: Sep. 17, 2023. [Online]. Available: https://link.springer.com/10.1007/978-3-319-20125-2_14.

[57] X. Du, C. Hargreaves, J. Sheppard, and M. Scanlon, "TraceGen: User activity emulation for digital forensic test image generation," *Forensic Science International: Digital Investigation*, vol. 38, p. 301 133, Oct. 1, 2021, ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2021.301133. Accessed: Dec. 30, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721000317.

[58] T. Göbel, T. Schäfer, J. Hachenberger, J. Türr, and H. Baier, "A Novel Approach for Generating Synthetic Datasets for Digital Forensics," in *Advances in Digital Forensics XVI*, G. Peterson and S. Shenoi, Eds., vol. 589, Cham: Springer International Publishing, 2020, pp. 73–93, ISBN: 978-3-030-56222-9 978-3-030-56223-6. DOI: 10.1007/978-3-030-56223-6_5. Accessed: Sep. 17, 2023. [Online]. Available: http://link.springer.com/10.1007/978-3-030-56223-6_5.

[59] M. Demmel, T. Göbel, P. Gonçalves, and H. Baier, "Data Synthesis Is Going Mobile—On Community-Driven Dataset Generation for Android Devices," *Digital Threats*, vol. 5, no. 3, 30:1–30:19, Oct. 26, 2024. DOI: 10.1145/3688807. Accessed: Dec. 29, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3688807.

[60] *Astral-sh/uv*, Astral, Mar. 8, 2025. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/astral-sh/uv.

[61] *PyCQA/flake8*, Python Code Quality Authority, Mar. 6, 2025. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/PyCQA/flake8.

[62] Ł. Langa and contributors to Black, *Black: The uncompromising Python code formatter*, Mar. 8, 2025. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/psf/black.

[63] *PyCQA/isort*, Python Code Quality Authority, Mar. 7, 2025. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/PyCQA/isort.

[64] *Python/mypy*, Python, Mar. 8, 2025. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/python/mypy.

[65] Jjk422, *Jjk422/ForGen*, Aug. 15, 2019. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/Jjk422/ForGen.

[66] S. M. Larson, *Sethmlarson/virtualbox-python*, Jan. 16, 2025. Accessed: Feb. 10, 2025. [Online]. Available: https://github.com/sethmlarson/virtualbox-python.

[67] *SeleniumHQ/selenium*, Selenium, Mar. 5, 2025. Accessed: Mar. 5, 2025. [Online]. Available: https://github.com/SeleniumHQ/selenium.

[68] *Microsoft/playwright-python*, Microsoft, Feb. 5, 2025. Accessed: Feb. 5, 2025. [Online]. Available: https://github.com/microsoft/playwright-python.

[69] A. Sweigart, *Asweigart/pyautogui*, Mar. 5, 2025. Accessed: Mar. 5, 2025. [Online]. Available: https://github.com/asweigart/pyautogui.

[70] *Tomerfiliba-org/rpyc*, tomerfiliba-org, Feb. 5, 2025. Accessed: Feb. 5, 2025. [Online]. Available: https://github.com/tomerfiliba-org/rpyc.

[71] *Pyinstaller/pyinstaller*, PyInstaller, Feb. 21, 2025. Accessed: Feb. 22, 2025. [Online]. Available: https://github.com/pyinstaller/pyinstaller.

[72] "Theory of Operation — RPyC," Accessed: Feb. 6, 2025. [Online]. Available: https://rpyc.readthedocs.io/en/latest/docs/theory.html#theory.

[73] A. Neyaz, N. Shashidhar, and U. Karabiyik, "Forensic Analysis of Wear Leveling on Solid-State Media," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, Aug. 2018, pp. 1706–1710. DOI: 10.1109/TrustCom/BigDataSE.2018.00256. Accessed: Feb. 5, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8456124.

[74] A. Pshenichkin. "So you think you want to write a deterministic hypervisor?" Antithesis, Accessed: Feb. 22, 2025. [Online]. Available: https://antithesis.com/blog/deterministic_hypervisor/.

[75] *PyFilesystem/pyfilesystem2*, pyFilesystem, Mar. 4, 2025. Accessed: Mar. 5, 2025. [Online]. Available: https://github.com/PyFilesystem/pyfilesystem2.

[76] C. Lalancette, *Clalancette/pycdlib*, Feb. 6, 2025. Accessed: Feb. 23, 2025. [Online]. Available: https://github.com/clalancette/pycdlib.

[77] *Sleuthkit/sleuthkit*, The Sleuth Kit, Feb. 26, 2025. Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/sleuthkit/sleuthkit.

[78] *Py4n6/pytsk*, py4n6, Feb. 25, 2025. Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/py4n6/pytsk.

[79] *Libyal/libyal*, libyal, Feb. 26, 2025. Accessed: Mar. 5, 2025. [Online]. Available: https://github.com/libyal/libyal.

[80] *Log2timeline/dfvfs*, log2timeline, Feb. 12, 2025. Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/log2timeline/dfvfs.

[81] Exterro. "FTK Imager - Forensic Data Imaging and Preview Solution," Exterro, Accessed: Mar. 6, 2025. [Online]. Available: https://www.exterro.com/digital-forensics-software/ftk-imager.

[82] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 313–324. DOI: 10.1109/HPCA.2017.10. Accessed: Feb. 11, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7920835.

[83] Volatility Foundation, *Volatility 3*, Mar. 6, 2025. Accessed: Mar. 6, 2025. [Online]. Available: https://github.com/volatilityfoundation/volatility3.

[84] E. Casey, G. Back, and S. Barnum, "Leveraging CybOX™ to standardize representation and exchange of digital forensic information," *Digital Investigation*, DFRWS 2015 Europe, vol. 12, S102–S110, Mar. 1, 2015, ISSN: 1742-2876. DOI: 10.1016/j.diin.2015.01.014. Accessed: Dec. 3, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287615000158.

[85] J. Abbott, J. Bell, A. Clark, O. De Vel, and G. Mohay, "Automated recognition of event scenarios for digital forensics," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06, New York, NY, USA: Association for Computing Machinery, Apr. 23, 2006, pp. 293–300, ISBN: 978-1-59593-108-5. DOI: 10.1145/1141277.1141346. Accessed: Aug. 30, 2023. [Online]. Available: https://doi.org/10.1145/1141277.1141346.

[86] K. Conlan, I. Baggili, and F. Breitinger, "Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy," *Digital Investigation*, vol. 18, S66–S75, Aug. 7, 2016, ISSN: 1742-2876. DOI: 10.1016/j.diin.2016.04.006. Accessed: Nov. 27, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287616300378.

[87] "Casework/CASE-Mapping-Python," Accessed: Dec. 14, 2024. [Online]. Available: https://github.com/casework/CASE-Mapping-Python.

[88] *ucoProject/UCO:develop-2.0.0*, ucoProject, Feb. 16, 2025. Accessed: Mar. 6, 2025. [Online]. Available: https://github.com/ucoProject/UCO/tree/develop-2.0.0.

[89] S. Colvin et al., *Pydantic*, version v2.10.3, Dec. 2024. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/pydantic/pydantic.

[90] J. MacFarlane, A. Krewinkel, and J. Rosenthal, *Pandoc*, Mar. 15, 2025. Accessed: Mar. 15, 2025. [Online]. Available: https://github.com/jgm/pandoc.

[91] P. Wagler, *Wandmalfarbe/pandoc-latex-template*, Mar. 14, 2025. Accessed: Mar. 15, 2025. [Online]. Available: https://github.com/Wandmalfarbe/pandoc-latex-template.

[92] "Requests 2.31.0 documentation," Accessed: Dec. 5, 2023. [Online]. Available: https://requests.readthedocs.io/en/latest/.

[93] *Hashicorp/vagrant*, HashiCorp, Feb. 22, 2025. Accessed: Feb. 23, 2025. [Online]. Available: https://github.com/hashicorp/vagrant.

[94] HashiCorp. "HashiCorp Cloud Platform," Vagrant Public Registry, Accessed: Feb. 23, 2025. [Online]. Available: https://portal.cloud.hashicorp.com/vagrant/discover.

[95] R. A. Finkel, *Advanced Programming Language Design*. Addison-Wesley Reading, 1996.

[96] *Ansible/ansible*, Ansible, Mar. 8, 2025. Accessed: Mar. 8, 2025. [Online]. Available: https://github.com/ansible/ansible.

[97] Ansible project contributors. "Ansible playbooks," Ansible Community Documentation, Accessed: Feb. 24, 2025. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html.

[98] OpenAI. "Introducing Operator," OpenAI, Accessed: Feb. 10, 2025. [Online]. Available: https://openai.com/index/introducing-operator/.

[99] OpenAI. "Computer-Using Agent," OpenAI, Accessed: Feb. 10, 2025. [Online]. Available: https://openai.com/index/computer-using-agent/.

[100] S. Zhou et al. "WebArena: A Realistic Web Environment for Building Autonomous Agents." arXiv: 2307.13854 [cs], Accessed: Feb. 10, 2025. [Online]. Available: http://arxiv.org/abs/2307.13854, pre-published.

[101] H. He et al. "WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models." arXiv: 2401.13919 [cs], Accessed: Feb. 10, 2025. [Online]. Available: http://arxiv.org/abs/2401.13919, pre-published.

[102] T. Xie et al. "OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments." arXiv: 2404.07972 [cs], Accessed: Feb. 10, 2025. [Online]. Available: http://arxiv.org/abs/2404.07972, pre-published.

[103] M. Astekin, M. Hort, and L. Moonen, "An Exploratory Study on How Non-Determinism in Large Language Models Affects Log Parsing," in *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*, ser. InteNSE '24, New York, NY, USA: Association for Computing Machinery, Aug. 7, 2024, pp. 13–18, ISBN: 979-8-4007-0564-9. DOI: 10.1145/3643661.3643952. Accessed: Feb. 10, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3643661.3643952.

[104] Y. Song, G. Wang, S. Li, and B. Y. Lin. "The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism." arXiv: 2407.10457 [cs], Accessed: Feb. 10, 2025. [Online]. Available: http://arxiv.org/abs/2407.10457, pre-published.

[105] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An Empirical Study of the Non-Determinism of ChatGPT in Code Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, 42:1–42:28, Jan. 22, 2025, ISSN: 1049-331X. DOI: 10.1145/3697010. Accessed: Feb. 10, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3697010.

[106] J. Hwang, S. Zeng, F. y Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013, pp. 269–276. Accessed: Feb. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/6572995.

[107] *Vmware/pyvmomi*, VMware, Feb. 7, 2025. Accessed: Feb. 10, 2025. [Online]. Available: https://github.com/vmware/pyvmomi.

[108] M. Chernyshev, S. Zeadally, Z. Baig, and A. Woodward, "Mobile Forensics: Advances, Challenges, and Research Opportunities," *IEEE Security & Privacy*, vol. 15, no. 6, pp. 42–51, Nov. 2017, ISSN: 1558-4046. DOI: 10.1109/MSP.2017.4251107. Accessed: Feb. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8123468.

[109] T. Sutikno and I. Busthomi, "Capabilities of cellebrite universal forensics extraction device in mobile device forensics," *Computer Science and Information Technologies*, vol. 5, no. 3, pp. 254–264, 3 Nov. 1, 2024, ISSN: 2722-3221. DOI: 10.11591/csit.v5i3.p254-264. Accessed: Feb. 10, 2025. [Online]. Available: https://www.iaesprime.com/index.php/csit/article/view/459.

[110] P. Gonçalves, K. Dološ, M. Stebner, A. Attenberger, and H. Baier, "Revisiting the dataset gap problem – On availability, assessment and perspective of mobile forensic corpora," *Forensic Science International: Digital Investigation*, vol. 43, p. 301 439, Sep. 1, 2022, ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2022.301439. Accessed: Feb. 10, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281722001202.

[111] A. A. Ceballos Delgado, W. B. Glisson, G. Grispos, and K.-K. R. Choo, "FADE: A forensic image generator for android device education," *WIREs Forensic Science*, vol. 4, no. 2, e1432, 2022, ISSN: 2573-9468. DOI: 10.1002/wfs2.1432. Accessed: Feb. 13, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/wfs2.1432.

[112] M. Janicki, M. Katara, and T. Pääkkönen, "Obstacles and opportunities in deploying model-based GUI testing of mobile software: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 313–341, 2012, ISSN: 1099-1689. DOI: 10.1002/stvr.460. Accessed: Dec. 30, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.460.

[113] L. Nagowah and G. Sowamber, "A novel approach of automation testing on mobile devices," in *2012 International Conference on Computer & Information Science (ICCIS)*, vol. 2, Jun. 2012, pp. 924–930. DOI: 10.1109/ICCISci.2012.6297158. Accessed: Feb. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/6297158.

[114] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do Developers Test Android Applications?" In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 613–622. DOI: 10.1109/ICSME.2017.47. Accessed: Feb. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8094467.

[115] *ucoProject/UCO*, ucoProject, Feb. 16, 2025. Accessed: Mar. 9, 2025. [Online]. Available: https://github.com/ucoProject/UCO.

Appendix A

Architectural diagrams

Figure A.1 is a complete representation of the AKF architecture, excluding labels. The remaining diagrams in this appendix chapter focus on individual modules in greater detail; Figure A.1 is intended to help the reader understand the relationships between individual sub-diagrams, which contain the labels omitted from this figure.

The following conventions are used for all architectural figures throughout this thesis, including those below:

- Elements with a **solid border** refer to AKF submodules, which contain processing logic.
- Elements with a **dotted border** refer to inputs, such as scripts and images, that should be used to construct a dataset.
- Elements with a **dashed border** refer to outputs, such as disk images and memory dumps, that should be included with a complete dataset.

In general, colored boxes refer to a distinct module. The sole exception is the *translation unit* (colored green), which is distinctly colored from other submodules due to its complexity.

The action automation library, described throughout chapter 4, is depicted in Figure A.2.

The virtualized environment and its agent, also described throughout chapter 4, is depicted in Figure A.3. Note that some elements depicted in this figure are described in chapter 5 and chapter 6.

The output and validation library, described throughout chapter 5, is depicted in Figure A.4. Note that this is distinct from Figure 5.2, which includes various virtual machine outputs due to their role in the chapter's discussion.

Finally, modules related to scenario construction, as described in chapter 6, are depicted in Figure A.5.

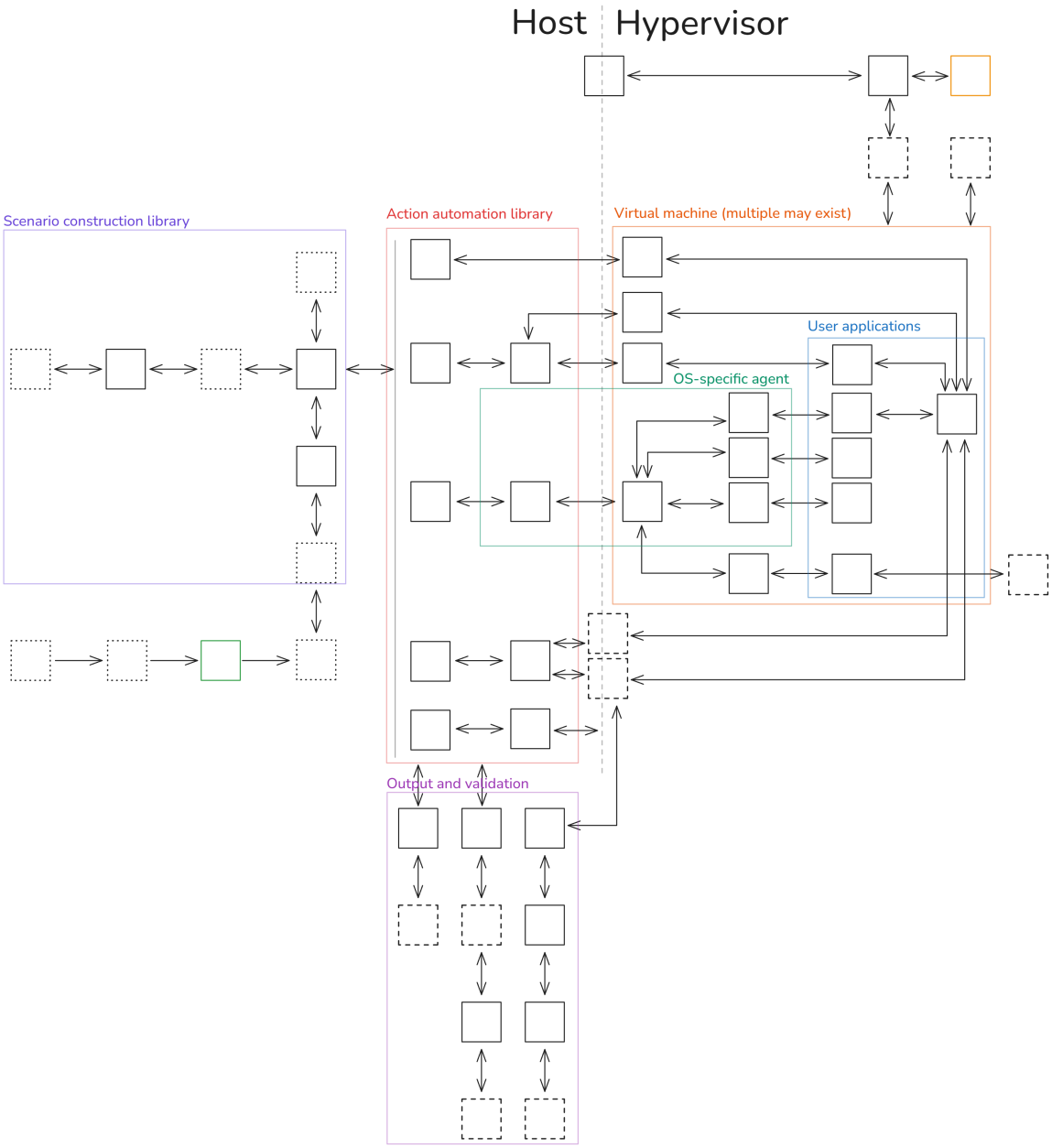

Figure A.1: Complete diagram of AKF modules without labels

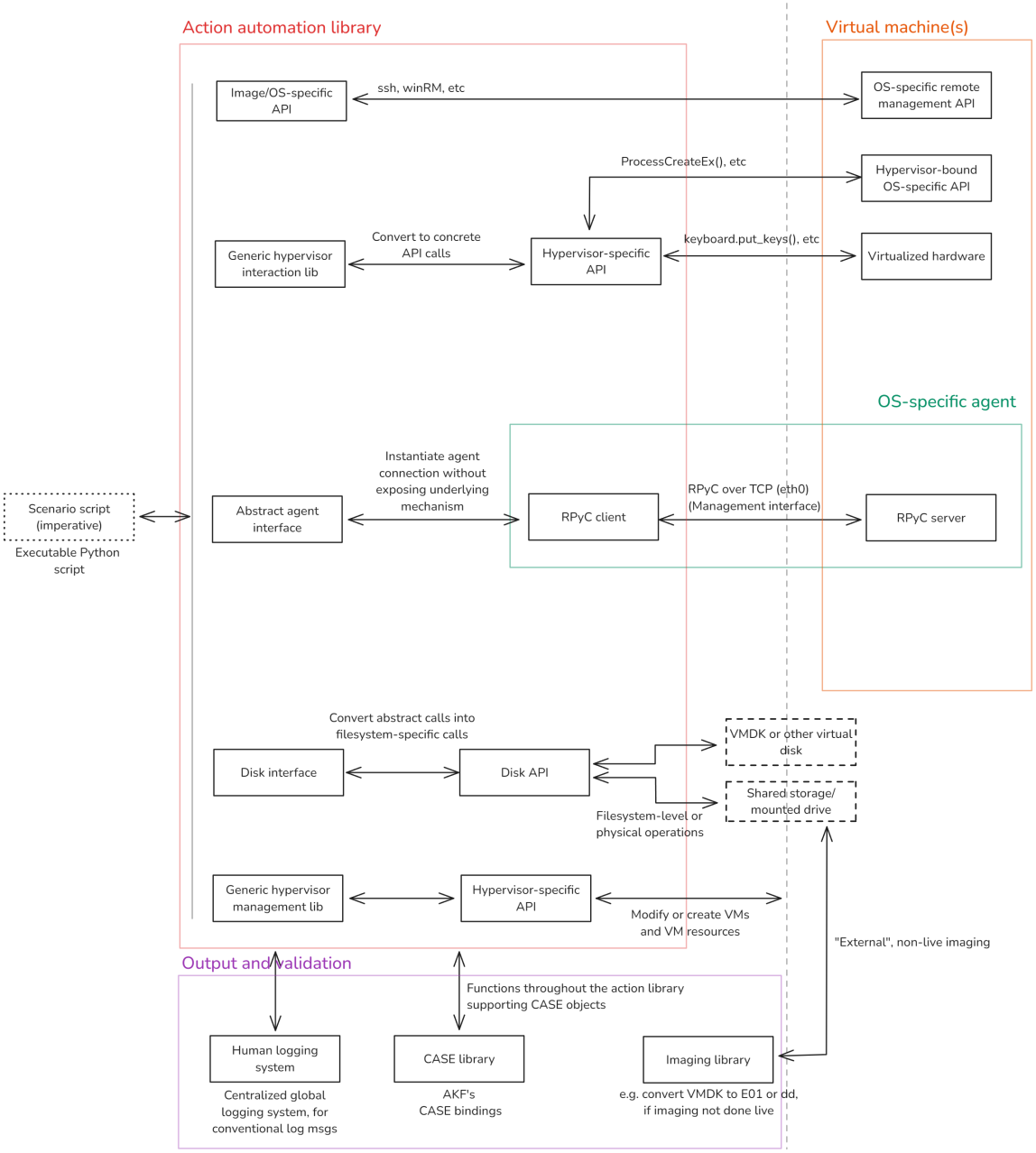

Figure A.2: Detailed diagram of `akflib` and related modules

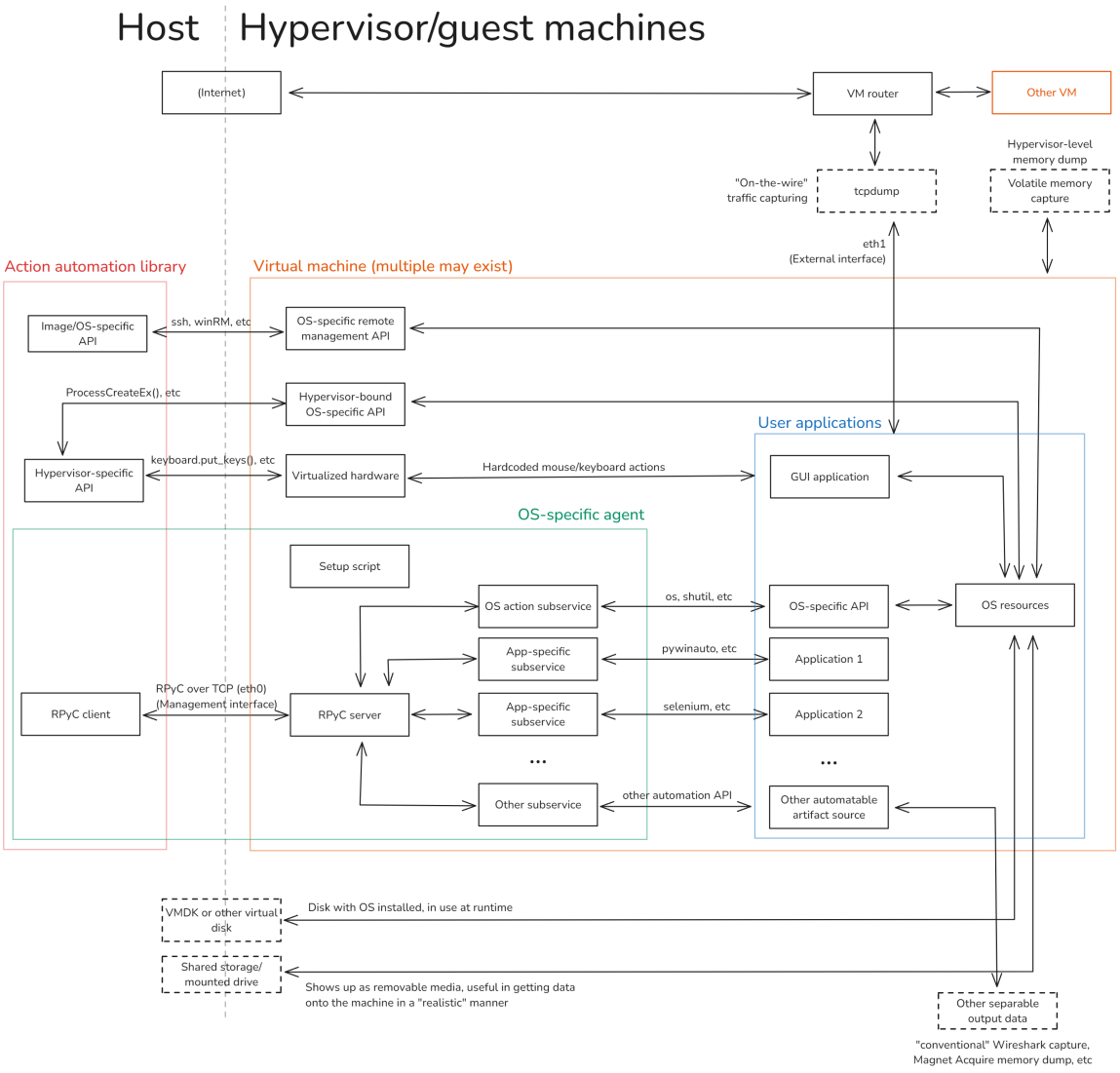

Figure A.3: Detailed diagram of AKF's virtualized environment

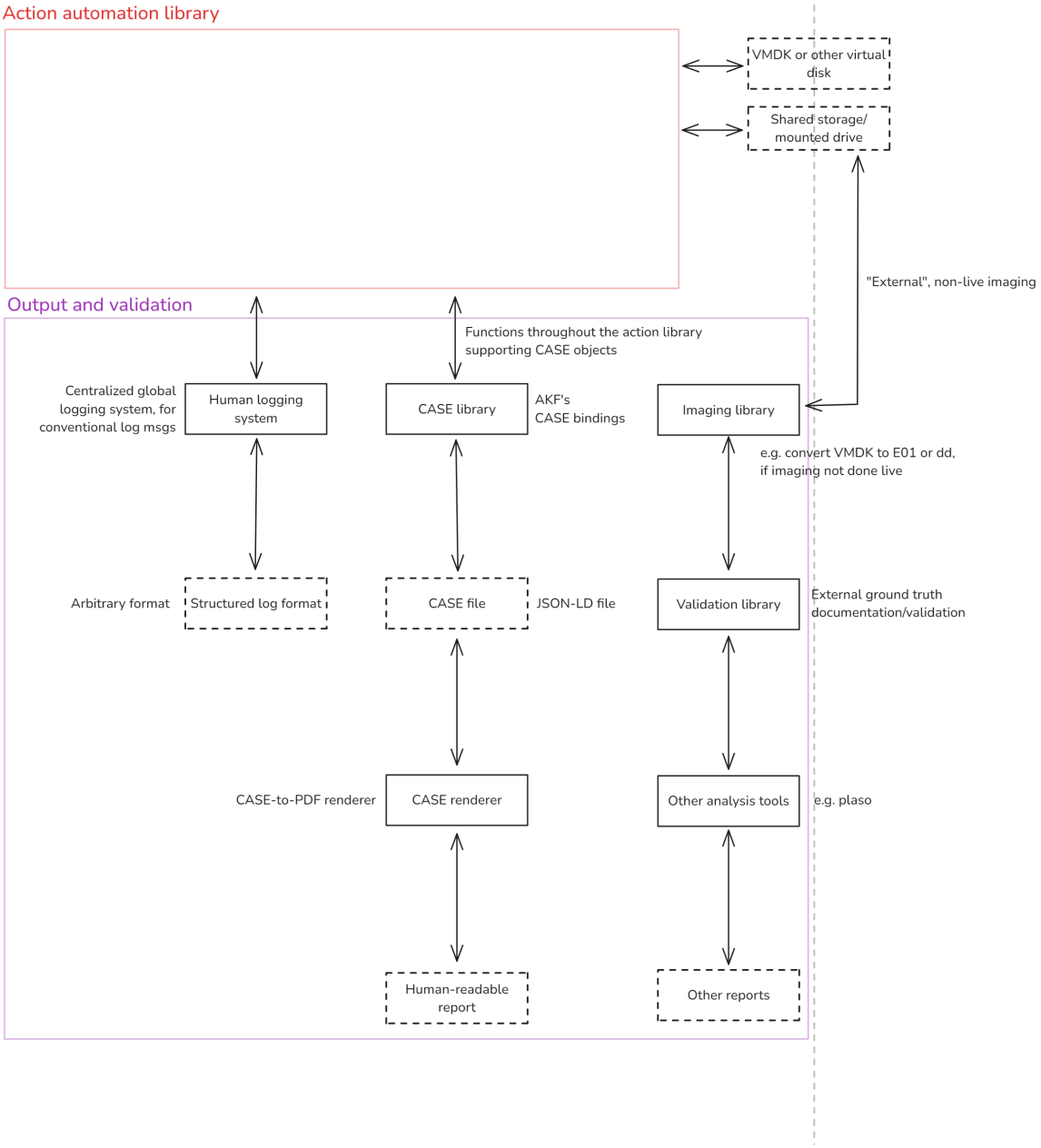

Figure A.4: Detailed diagram of AKF modules responsible for output and validation

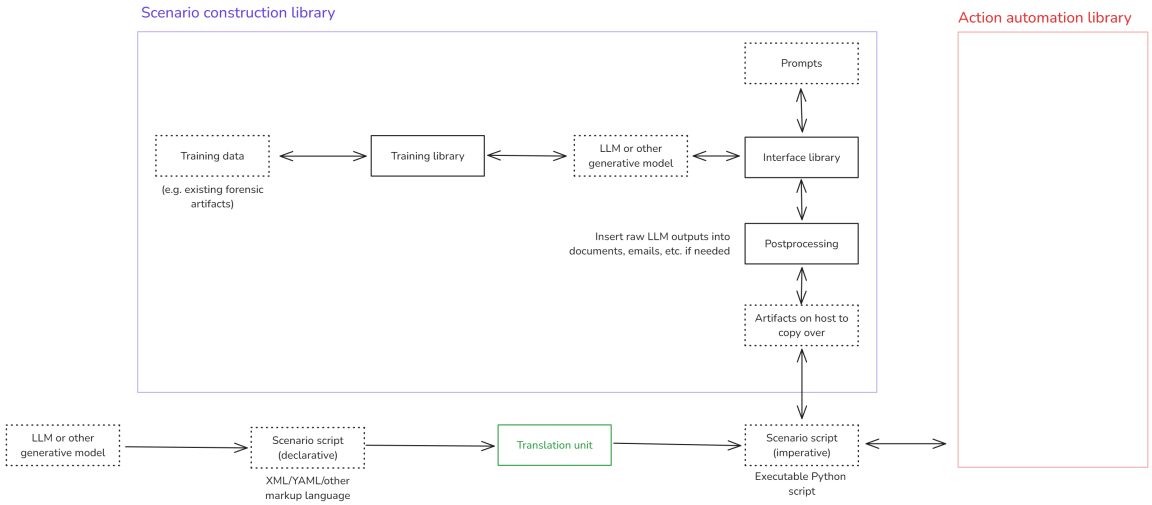

Figure A.5: Detailed diagram of AKF modules related to scenario construction

Appendix B

Code samples

B.1 Code repositories

The complete implementation of AKF is available on GitHub at the following locations:

- Core libraries (`akflib`): https://github.com/lgactna/akflib
- The AKF agent for Windows: https://github.com/lgactna/akf-windows
- Standalone CASE/UCO 2.0 bindings for Python: https://github.com/lgactna/CASE-pydantic

B.2 Comparison of ForTrace and AKF agents

Invoking commands in ForTrace [45] is straightforward; at the network level, commands are performed by issuing a simple string to the agent running on the target VM. For example, invoking a command to add a new user to the VM can be seen in Listing B.1:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```

# From Demo.py

guest = virtual_machine_monitor1.create_guest(guest_name=imagename,
    platform="windows")

# Wait for the VM to connect to the VMM

guest.waitTillAgentIsConnected()

# create userManagement object

userManagement_obj = guest.application("userManagement", {})

# Add different users

logger.info("Adding user1")
try:
    userManagement_obj.addUser("user1", "password")
    while userManagement_obj.is_busy is True:
        time.sleep(1)
except Exception as e:
    print("An error occured: ")
    print(e)
time.sleep(5)
```

Listing B.1: Creating a new user through ForTrace agent commands

The agent's main loop, which receives and interprets these commands, is also straightforward. A simplified version of the main loop is depicted in Listing B.2:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
def main():
    # create logger
    logger = create_logger('guestAgent', logging.INFO)

    logger.info("create Agent")
    a = Agent(operating_system=platform.system().lower(), logger=logger)
    logger.info("connect to fortrace controller: %s:%i" % (
    fortrace_CONTROLLER_IP, fortrace_CONTROLLER_PORT))
    a.connect(fortrace_CONTROLLER_IP, fortrace_CONTROLLER_PORT)

    # let all network interfaces come up
    time.sleep(15)

    # inform fortrace controller about network configuration
    a.register()

    # wait for commands
    while 1:
```

```
18          time.sleep(1)
19          a.receiveCommands()
```

Listing B.2: Simplified ForTrace agent entry point

When the `Agent` is instantiated, it binds a TCP socket to the configured port and IP address, where it expects the server (the VMM) to issue commands. `Agent.receiveCommands()` executes commands by reading the socket, parsing the received content, and then invoking `Agent.do_command()`, which converts the command message into a specific Python function call with arguments.

The server, or "virtual machine monitor" (VMM), issues commands over TCP to a running instance of the agent on the VM. Each module under `fortrace.application` can be thought of as a coherent group of commands associated with a particular user application (such as Firefox or Thunderbird). Recall from Listing B.1 how we found our `userManagement` application and invoked `addUser`; a simplified snippet is depicted in Listing B.3:

```
    Extension = .otf ,
    UprightFont = Inconsolatazi4-Regular,
    BoldFont = Inconsolatazi4-Bold
1 userManagement_obj = guest.application("userManagement", {})
2 userManagement_obj.addUser("user1", "password")
```

Listing B.3: Minimal ForTrace application API usage

At a high level, the `application()` call attempts to import `fortrace.application.{application_name}`. Although not enforced by a higher-level interface, each of these modules contains subclasses of the following four classes (defined in `fortrace.application.application`) at minimum, with additional helper classes for OS-specific functionality or other modularity as needed:

- `ApplicationVmmSide`: Contains one function for each command implemented in `ApplicationGuestSide`, building a message that will be interpreted and acted upon by the corresponding `ApplicationGuestSideCommands` class.
- `ApplicationVmmSideCommands`: Accepts and interprets module-specific messages re-

turned by the agent, which may be used to update the remote state as tracked by the host.

- `ApplicationGuestSide`: Implements the actual application-specific functionality for the agent, providing one function for each separated command by this module.
- `ApplicationGuestSideCommands`: Interprets commands and arguments, calling the respective function in the corresponding `ApplicationGuestSide` subclass. This allows the actual dispatch of commands to be delegated to this class, which is free to choose how actions are performed (such as the use of threading and multiprocessing) as well as any module-wide state it may need to maintain.

This naming convention is intentional, as individual modules dictate the name of the module in camelcase. For example, `fortrace.application.userManagement` contains `UserManagementVmmSide`, `UserManagementVmmSideCommands`, and so on. Discovering and getting handles to these classes is performed through string manipulation of the relevant application's module name, as shown in Listing B.4 (an abridged version of the `Agent.do_command()` method):

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
1  def do_command(self, command):
2      com = command.split(" ")
3      package = com[0]
4      module = com[1]
5
6      # load class moduleGuestSide and moduleGuestSideCommands
7      name = "fortrace." + package + "." + module
8      self.logger.debug("module to load: " + name)
9      mod = __import__(name, fromlist=[''])
10     self.logger.debug("module '" + module + "' will be loaded via
       __import__")
11     class_commands = getattr(mod, module[0].upper() + module[1:] + '
       GuestSideCommands')
12
```

```
13      class_commands.commands(self, app_obj, " ".join(com[1:]))
```

Listing B.4: Demonstration of ForTrace agent module discovery and command execution

In the example above, `UserManagementGuestSide.addUser()` contains the code to execute on the guest when `addUser()` is called, such as adding the registry keys needed for a user to be created. On the other hand, `UserManagementVmmSide.addUser()` contains the code to send a message over TCP that the agent will understand, eventually leading to the execution of the agent's version of `addUser()`.

More precisely, the `ApplicationVmmSide` subclass effectively serves as the API for calling the associated functions in the `ApplicationGuestSide` class running on the virtual machine. This subclass, along with the complete agent-side code, is stored in a single file. For example, the API and code for opening a Firefox browser window can be seen in Listing B.5:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
1  class WebBrowserFirefoxVmmSide(ApplicationVmmSide):
2      def open(self, url):
3          """Sends a command to open a webBrowserFirefox on the associated
       guest.
4
5          @param url: Website to open.
6          """
7          try:
8              self.logger.info("function: WebBrowserFirefoxVmmSide::open")
9              self.url = url
10             self.window_id = self.guest_obj.current_window_id
11             self.guest_obj.send(
12                 "application " + "webBrowserFirefox " + str(self.
       window_id) + " open " + self.webBrowserFirefox + " " + self.url)
13
14             self.guest_obj.current_window_id += 1
15         except Exception as e:
16             raise Exception("error WebBrowserFirefoxVmmSide::open: " +
       str(e))
17
18 # Example usage:
19
20 browser = WebBrowserFirefoxVmmSide()
```

```
21 browser.open("google.com")
```

Listing B.5: Sample ForTrace agent API implementation

Calling the `open()` command from the host sends a structured message to the agent, generally of the form seen in Listing B.6:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
1 application webBrowserFirefox <window id> open Firefox <url>
```

Listing B.6: Sample ForTrace agent protocol message

Upon receiving this message, the agent's main loop will search for the `webBrowserFirefox` module and import its corresponding `ApplicationGuestSideCommands` subclass. After any state management, the subclass will then search for a function called `open` in its corresponding `ApplicationGuestSide` class, as implemented in Listing B.7:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
1  class WebBrowserFirefoxGuestSide(ApplicationGuestSide):
2      def open(self, args):
3          # docstring omitted
4          try:
5              arguments = args.split(" ")
6              web_browser = arguments[0]
7              url = arguments[1]
8
9              if len(arguments) > 2:
10                 self.timeout = arguments[2]
11             else:
12                 self.timeout = 30
13
14             self.logger.info(self.module_name + "GuestSide::open")
15             self.last_driven_url = url
16             self.logger.debug("URL to call: " + url)
17
18             self.logger.info("open url: " + url)
19
20             self.helper.run_firefox()  # start ff session
21
22             retval = self.helper.navigate_to_url(url)  # browse to the
    specified url
23             if not retval:
24                 self.logger.warning("could not open url")
25
```

```
26              self.agent_object.send("application " + self.module_name + "
    " + str(self.window_id) + " opened")
27
28              self.agent_object.send("application " + self.module_name + "
    " + str(self.window_id) + " ready")
29              self.window_is_crushed = False
30          except Exception as e:
31              # Some logging/teardown...
32              self.window_is_crushed = True
33              self.agent_object.send("application " + self.module_name + "
    " + str(self.window_id) + " error")
```

Listing B.7: Corresponding ForTrace agent-side call

As described in subsection 4.2.2, this is achieved in AKF using RPyC services, which are analogous to the `ApplicationGuestSideCommands` and `ApplicationGuestSide` classes of individual ForTrace modules. In addition to providing the RPyC services themselves, agents also implement an API to call these services using typed functions.

For example, suppose we wanted to implement functionality similar to the ForTrace code above, allowing users to navigate to a webpage. An equivalent RPyC service and its corresponding API may be implemented as seen in Listing B.8:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
1
2  # Server-side code
3
4  class ChromiumService(AKFService):
5      def exposed_set_browser(
6          self, browser_type: Literal["msedge", "chrome"], profile: str =
       "Default"
7      ) -> BrowserContext:
8          # Various setup code...
9          self.browser = chromium.launch_persistent_context(
10             headless=False,
11             user_data_dir=profile_path,
12             channel=browser_type,
13             args=[f"--profile-directory={profile}"],
14         )
15
16         return self.browser
17
18 # Client-side API
19
20 class ChromiumServiceAPI(WindowsServiceAPI):
21     def __init__(self, host: str, port: int) -> None:
```

```
        """
        Initialize the API with an RPyC connection to the service.
        """
        self.rpyc_conn = rpyc.connect(
            host,
            port,
            config={"sync_request_timeout": None},
        )

    def set_browser(
        self, browser_type: Literal["msedge", "chrome"], profile: str = "Default"
    ) -> BrowserContext:
        self.browser = self.rpyc_conn.root.set_browser(browser_type, profile)
        return self.browser
```

Listing B.8: Minimal reimplementation of ForTrace module as an RPyC service

Implementing support for a particular service on the client-side API is as simple as calling an untyped function `rpyc_conn.root.set_browser()`. By wrapping it around a typed function, `ChromiumServiceAPI.set_browser()`, users regain the ability to use code completion and static linting tools.

B.3 CASE Python bindings

As described in subsection 5.2.1, CASE is defined using Turtle, allowing objects to be written in a human-readable text format. For example, the following set of triples in Listing B.9 describes an object called `ApplicationFacet` with two properties, `numberOfLaunches` and `applicationIdentifier`:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
observable:ApplicationFacet
    a
        owl:Class ,
        sh:NodeShape
        ;
    rdfs:subClassOf core:Facet ;
    rdfs:label "ApplicationFacet"@en ;
    rdfs:comment "An application facet is a grouping of characteristics
    unique to a particular software program designed for end users."@en
```

```
    ;
    sh:property
        [
            sh:datatype xsd:integer ;
            sh:maxCount "1"^^xsd:integer ;
            sh:nodeKind sh:Literal ;
            sh:path observable:numberOfLaunches ;
        ] ,
        [
            sh:datatype xsd:string ;
            sh:maxCount "1"^^xsd:integer ;
            sh:nodeKind sh:Literal ;
            sh:path observable:applicationIdentifier ;
        ] ,
        ...
        ;
    sh:targetClass observable:ApplicationFacet ;
    .
```

Listing B.9: Example CASE object definition for applications [115]

An instance of an `Application` object may thus be represented in the JSON-LD format using the `ApplicationFacet` as seen in Listing B.10, including some attributes omitted from the example in Listing B.9:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
{
    "@id": "kb:dcec8d09-a8bc-4b7c-93ab-16c7b363d48b",
    "@type": "uco-observable:Application",
    "uco-core:hasFacet": [
        {
            "@id": "kb:68004de9-1139-405f-aea7-2c05f3a84709",
            "@type": "uco-observable:ApplicationFacet",
            "uco-observable:numberOfLaunches": 12,
            "uco-observable:applicationIdentifier": "test"
        },
    ]
}
```

Listing B.10: Instantiated CASE application object as JSON-LD

The CASE project provides official Python bindings [87]. One notable implementation detail is that it stores all object attributes in a dictionary after instantiation. This can be seen in the example of an `ApplicationFacet` in Listing B.11 below. Intuitive usage suggests that the `application_identifier` attribute is accessible through the

facet object, but it must instead be accessed as a dictionary key with a non-intuitive name.

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
facet = ApplicationFacet(application_identifier = "test",
    number_of_launches=3)

# These attributes do not exist

facet.application_identifier
facet.number_of_launches

# The number of launches must be accessed as a dictionary key, which
    does

# not return a simple integer, but instead returns a dictionary
app_facet['uco-observable:numberOfLaunches']

# -> {"@type": "xsd:integer", "@value": "3"}
```

Listing B.11: Sample usage of official CASE Python bindings [87]

AKF's Pydantic-based bindings greatly simplify the declaration of individual CASE objects while allowing typical attribute-based access. For example, AKF's declaration of ApplicationFacet is shown in Listing B.12:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
from typing import Optional

from uco import core

class ApplicationFacet(core.Facet):
    numberOfLaunches: Optional[int] = None
    applicationIdentifier: Optional[str] = None
```

Listing B.12: Example of Pydantic-based CASE object declaration used by AKF

This definition is only three lines, which is 32 lines shorter than the declaration of ApplicationFacet provided by the CASE project's existing Python bindings (excluding the docstring).

A simple CASE bundle, representing the complete contents of a forensic scenario, can be constructed and exported to JSON-LD using AKF libraries as shown in List-

ing B.13:

```
    Extension = .otf ,
    UprightFont = Inconsolatazi4-Regular,
    BoldFont = Inconsolatazi4-Bold
from caselib import case, uco

bundle = uco.core.Bundle(
    description="An Example Case File",
    specVersion="UCO/CASE 2.0",
    tag="Sample artifacts",
)

url_object = uco.observable.ObservableObject()
url_facet = uco.observable.URLFacet(fullValue="www.docker.com/howto")
url_object.hasFacet.append(url_facet)
bundle.object.append(url_object)

with open("example.jsonld", "wt+") as f:
    data = bundle.model_dump(serialize_as_any=True)
    f.write(json.dumps(data, indent=2))
```

Listing B.13: Demonstration of bundle serialization using the AKF Python bindings for CASE

Although the UCO/CASE 2.0 Python bindings developed as part of AKF significantly improve usability and flexibility over the existing CASE bindings, several limitations of the library do not make it fully compliant with the CASE ontology.

In particular, the Python bindings aim to keep object definitions as simple as possible, preferring native Python types where possible. This means that some information from the ontology is lost when converting them to their respective Python class. Several examples include:

- **Datatypes**: Many XSD datatypes are not equivalent between the turtle files and Python bindings. For example, the arbitrary-precision `xsd:decimal` type is represented as a native Python float, but is also written out as a native JSON float on serialization to JSON-LD. The resulting datatype is `xsd:float`, which may cause information to be lost.
- **Vocabularies**: Many CASE "vocabularies", a datatype in which a field's value should be chosen from a fixed set of values, are correctly converted to native

Python string enumerations. However, the vocabulary datatype is not serialized, only the value; thus, the resulting JSON-LD makes no indication that the selected value is actually from the vocabulary datatype, even if the value itself is inside the vocabulary set. For example, if the string "MD5" is a member of the `HashNameVocab` datatype, our Python bindings serialize this as a standard `xsd:string`, not `vocabulary:HashNameVocab`.

- **Field names**: Some fields of CASE objects, such as the `from` field of an email message, are reserved Python keywords that may not be used as the name of a variable. To solve this, we append an underscore to any fields that would violate this rule when automatically generating Pydantic classes from the Turtle RDF files. However, the original name of the field is not preserved when serializing objects; although the simplest fix is to remove trailing underscores when serializing, a more robust solution may be to attach the "serialized name" to the Pydantic field.
- **Dangling references**: There are no built-in mechanisms to ensure that an object is serialized in its "full" form at least once; that is, a user could place references to an object identifier throughout the document without adding the object being referred to.

Additionally, one major feature of CASE/UCO is not currently supported. CASE allows objects to be of multiple types at once, such as a disk image marked as both an `observable:File` and an `observable:Image`. Python types can inherit from multiple types, but a single object may not be multiple types at once; it would be necessary to create a "wrapper" type that encompasses both types and correctly serializes these types as expected.

B.4 Historical declarative syntaxes

While designing the declarative syntax for AKF, the syntaxes ofD-FET [54], SFX [46], and Yannikos et al. [42] were reviewed. For completeness, brief examples of scenarios in each of these synthesizers are included here. Notably, none of these works provide details on how the parser of their declarative languages is implemented. Similarly, few details are provided about the architecture and design of the code used to carry out actions based on interpreted declarative instructions. This lack of detail may be attributed to the fact that the scenario creation enabled by these declarative syntaxes, rather than the syntaxes themselves, was the primary focus of these works.

D-FET uses a custom language that does not depend on any existing text-based languages [54], as seen in the simple scenario in Listing B.14:

```
Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold
INSTANCE LOAD [Image=WINDOWS2003]
MOUNT INSTANCE [Disk=STANDARDDISK] AS [Partition="c"]
ACTIVITY LOAD [Number=12] [Type=JPEG IMAGES; Class=DRUGS]
    INTO [Folder=USER FOLDER]
    AT [Period=1 MINUTE] [Interval=INTERVAL]
    FOR [User=Fred]
```

Listing B.14: Sample D-FET declarative scenario without events [54]

The scenario above will create an instance based on the WINDOWS2003 image from the Host Forensics Image library (which contains pre-created images from OS installation discs). It will then load 12 JPEG images from the "DRUGS" class of images using the "STANDARDDISK" disk image. This appears to create a host with *predefined*, but not *timed* activity – the machine simply begins in this state rather than simulating a user doing this over some period of time.

To generate timed activity that can be placed on a timeline, D-FET allows users to add "events." The sequence of events in Listing B.15 directs the synthesizer to log in as a user, delete files, and log out.

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
INSTANCE LOAD [Image=WINDOWS2003]
MOUNT INSTANCE [Disk=STANDARDDISK] AS [Partition="c"]
ACTIVITY LOAD [Number=12] [Type=JPEG IMAGES; Class=DRUGS]
    INTO [Folder=USER FOLDER]
    AT [Period=1 MINUTE] [Interval=INTERVAL]
    FOR [User=Fred]
ACTIVITY EVENT [Event=LOGIN; User=Fred]
ACTIVITY EVENT [Event=DELETEFILE; User=Fred; File=JPEF IMAGES]
ACTIVITY EVENT [Event=LOGOUT; User=Fred ]
```

Listing B.15: Extension of the D-FET declarative scenario with events [54]

SFX uses an XML-based language with tags and attributes that are easily readable [46], as shown in Listing B.16:

Extension = .otf ,
UprightFont = Inconsolatazi4-Regular,
BoldFont = Inconsolatazi4-Bold

```
<disk size="512M" alignment="cylinder" diskid="0x12345678">
    <partition index="p1" hidden="0" size="48M" type="vfat">
        <expand archive="part1.zip" />
        <copy from="fake.dat" to="/fake01.dat" />
        <copy from="fake.dat" to="/fake02.dat" />
        <delete from="/Thomas.jpg" />
    </partition>
    <partition index="p2" hidden="0" size="48M" type="ntfs">
        <base os="windows7x64" />
        <copy from="fake.dat" to="/fake03.dat" />
        <copy from="fake.dat" to="/fake04.dat" />
        <user username="Gordon">
            <browserhistory browser="firefox">
                <url link="[http://bbc.co.uk"](http://bbc.co.uk") time="
    13:14:00 1 Jan 2013" />
            </browserhistory>
        </user>
    </partition>
    <partition index="p3" hidden="1" size="64M" type="ntfs">
        <expand archive="part3.zip" />
        <copy from="fake.dat" to="/fake11.dat" />
        <copy from="fake.dat" to="/fake12.dat" />
        <delete from="/docs/image.exe" />
        <slackspace offset="20" file="/tomas.gif" message="This is a
    secret message" />
    </partition>
    <partition index="s1" hidden="0" size="144M" type="ext3">
        <base os="fedora15x64" />
        <expand archive="part2.tar" />
        <copy from="fake.dat" to="/tmp/fake01.dat" />
        <copy from="fake.dat" to="/tmp/fake02.dat" />
```

```
30      </partition>
31  </disk>
```

Listing B.16: Sample SFX declarative scenario expressed as XML [46]

Here, a user can define multiple partitions on a single disk, each with a distinct filesystem that may or may not contain an underlying operating system. SFX allows users to generate artifacts in multiple ways, with each unique application- or OS-specific feature using a distinct XML element name. Artifact generation features include copying files to the guest machine in bulk, inserting files in the slack space of an existing file, and using browser artifacts.

Finally, the work of Yannikos et al. is particularly notable because it appears to be fully GUI-based, expecting users to visually construct Markov chains to define scenarios [42]. An example scenario from their publication can be seen in Figure Figure B.1:

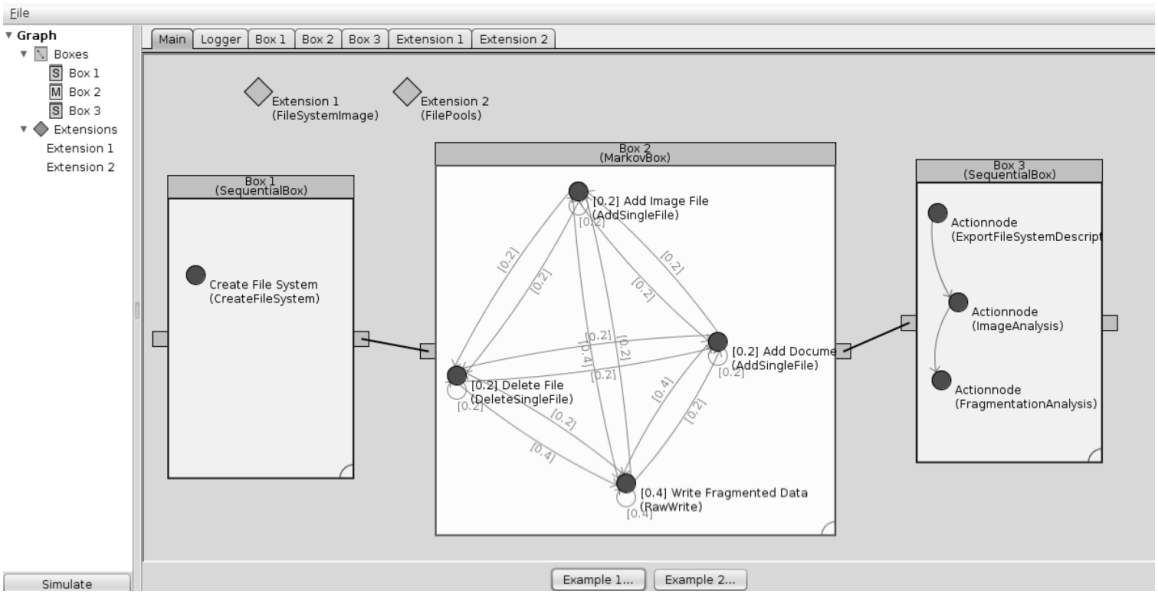

Figure B.1: GUI-based scenario declaration from Yannikos et al. [42]

Although details are relatively limited, each node appears to be a distinct action that can be automated. Individual nodes accept parameters that can be used to configure how their associated artifacts are created. Each "box" encompasses a

complete Markov chain, with transitions from one box to another occurring after an unknown condition is fulfilled. Finally, the diamonds outside the boxes, known as extensions, are libraries that provide extra functionality during the execution of the overall scenario.