Towards Scientific Incident Response

Jonathan M. Spring[1](✉) and David Pym[1,2]

[1] University College London, Gower Street, London, UK
jspring@cs.ucl.ac.uk, d.pym@ucl.ac.uk
[2] Alan Turing Institute, 96 Euston Road, London, UK

Abstract. A scientific incident analysis is one with a methodical, justifiable approach to the human decision-making process. Incident analysis is a good target for additional rigor because it is the most human-intensive part of incident response. Our goal is to provide the tools necessary for specifying precisely the reasoning process in incident analysis. Such tools are lacking, and are a necessary (though not sufficient) component of a more scientific analysis process. To reach this goal, we adapt tools from program verification that can capture and test abductive reasoning. As Charles Peirce coined the term in 1900, "Abduction is the process of forming an explanatory hypothesis. It is the only logical operation which introduces any new idea." We reference canonical examples as paradigms of decision-making during analysis. With these examples in mind, we design a logic capable of expressing decision-making during incident analysis. The result is that we can express, in machine-readable and precise language, the abductive hypotheses than an analyst makes, and the results of evaluating them. This result is beneficial because it opens up the opportunity of genuinely comparing analyst processes without revealing sensitive system details, as well as opening an opportunity towards improved decision-support via limited automation.

AQ1

AQ2

Keywords: Incident response · Digital forensics
Science of security · Mathematical modelling
Logical modelling · Intrusion analysis

1 Introduction and Motivation

Incident analysis is the central feature of incident response and digital forensics. Incident response and digital forensics overlap largely in their modes of analysis. Otherwise, they have different goals, and are done by different sorts of organizations. One might take a broad view of digital forensics and say it includes incident response, but realistically the term "digital forensics" has too many law-enforcement connotations for this broad usage to quite work. We focus on incident analysis, which, as defined by [28], includes the evidence collection, analysis, and reporting phases of our topic, whether that topic is incident response or digital forensic investigation.

Incident analysis is part of the study of what has occurred on computers and computer networks. An incident is an event that violates some security policy [26]; that policy may be but is not necessarily a law. We treat incident analysis as akin to scientific investigation. The analyst has a hypothetical model about how the incident occurred, and tests it by gathering evidence and adjusting the model based on the results. This is not the naive binary hypothesis testing of a high-school science lab. Rather, it is building models for a purpose based on empirical, structured observations of the world—the conception of science of security argued for by [29].

We inform our logic development with examples as well as the incident response standards review by [28]. For example, we draw inspiration from [30]. Since [30] is a description of an incident analyst tracking foreign spies through computer networks, it is a rather obvious paradigmatic case. Another, less obvious, example of incident analysis is Assistant for Randomized Monitoring Over Routes (ARMOR) [31]. ARMOR represents a kind of ongoing incident analysis, though of physical security. One reason [31] is relevant is that it is deployed decision-making. Although the form of our model is different, deployability is a major consideration of our design choices. Our working definition of incident analysis, adapting from that of 'investigation' in [6, p. 244], is:

> *Incident analysis*: a process by an *agent* to build a *model* and *explanation* of the *phenomenon* responsible for a security violation. The process is forensic (as distinguished from engineering or design which are forward-looking, though results should inform engineering). The process will include collection of *evidence*; discovery of interrelated *mechanisms*; investigative heuristics and *methodology*; and *reporting* results. Different incident analyses may have different *goals*, such as fixing the impacted system, attributing the attack, or legal prosecution.

Italicized terms may need their own definitions in future work. However, we are not seeking an ontology, and shall not elaborate them here.

Incident analysis is a key aspect of incident response. In turn, incident response is a crucial aspect of information security broadly. One essential aspect of infosec is feedback from incidents to 'preparation' and 'protection' [1].

The National Institute of Standards and Technology (NIST) guide on forensics in incident response recommends analysts use "a methodical approach" [14, p. 3-8]. However, nowhere does NIST provide such a methodical approach. This is a general failing. A recent review of published incident response documents and standards found that the literature lacked this middle-level of description [28]. Fine-grained, type-this-on-the-keyboard advice is available. And high-level, do-these-management-practices advice is available. But published guidance on a methodical approach to incident analysis is lacking, despite the central importance of the topic to cyber security.

We will contribute towards a methodical approach to incident analysis by building a logical language for analysts to document their reasoning process precisely. This contribution advances towards scientific incident analysis because it improves interpretation of evidence. A logic improves interpretation because it enables communication and repetition of the interpretive process, allowing for iterative improvement and collaboration.

To make use of our logic for improved decision-making, we also need to understand human cognition and how we think about thinking. [11] makes progress on this topic, applying the approach by [10] for reducing the impact of cognitive biases in analysis to computer network incident analysis. Our goal is to combine these aspects and provide logical tools such that steps of interpretation can be made explicit and the gaps in our knowledge identified more easily.

The paper continues as follows. Section 2 develops a new logic as a tool to express reasoning patterns within incident analysis. Section 3 demonstrates how to apply the logic by an example construction to express and elaborate the kill chain model from [12]. Section 4 lays out benefits to decision-making in security and future work.

2 Logic Definitions

In this section, we build a logical system as a tool for expressing paradigmatic features of incident analysis. These features include abductive hypothesis generation, evidence-based evaluation of hypothetical explanations, and reasoning about technical events. Section 3 will use the tools we build here to further elaborate our logic through an example.

A necessary part of a logical system is its model. A model, in this logic sense, is a mathematical structure with which we can interpret a proposition, and then determine whether it is satisfied or not. This sense is quite far from a scientific model. However, as [22] argues, a logic will be most effective when its logic model aligns with the salient features of a scientific model of the represented phenomenon. Therefore, we develop logical tools with the purpose of incident analysis in mind at every step. The phenomena of interest are violations of security policy; that is, a resultant state of a computer system. We will represent these as histories, composed of series of states of the computer.

We make a variety of choices to adapt the logic to incident analysis. Some are simple: incident analysis is largely about past events, so we include both past-tense and future-tense temporal operators. Others are more subtle. For example, we define a separation of network, storage, and processor resources at a basic level because practitioners think about, monitor, and defend these things quite differently. We wanted the logic to reflect this reality deeply. And some of our choices have an eye towards pragmatics of usability and deployable decision-making. As [18] describes, the road from formal logic to operational implementation is long. However, we include the 'and, separately' operator in our logic, which supports composable reasoning and an eye towards scalability.

2.1 Expressions

Our definition of expressions is essentially the same as [13] and [4]. An expression can be an integer, an atom, or a variable.

$$\begin{array}{rcl} E ::= & x & \text{Variable} \\ \mid & 37 & \text{Integer} \\ \mid & \texttt{nil} & \text{nil} \\ \mid & \texttt{a} & \text{atom} \\ \mid & \ldots & \end{array}$$

The open-ended definition of expressions allows for additional expressions so long as they can be interpreted in the semantic domain specified.

Our semantic domains are values, addresses, and content, analogous to and slightly more general than the values, stacks, and heaps used in [4]:

$$Val = Int \cup Atoms \cup Loc \quad A = Var \rightharpoonup_{fin} Val \quad C = Loc \rightharpoonup_{fin} Val \times Val$$

where $Loc = \{\ell, \ldots\}$ is an infinite set of locations, the term $Var = \{x, y, \ldots\}$ is a set of variables, $Atoms = \{\texttt{nil}, \texttt{a}, \ldots\}$ is the set of atoms, and finite partial functions are represented by $\rightharpoonup_{fin}$. Elements of addresses and content are $a \in A$ and $c \in C$, respectively. As is customary for stack variables, we do not provide an explicit operation for allocating address variables.

The domain of an element of addresses is $dom\,(a)$ for $a \in A$. Similarly, $dom\,(c)$ is the domain for an element of contents. Note that English grammar here may be confusing. An address a is a set of mappings from variables to values, not a singleton. Likewise, c is a set of content mappings, not a singleton.

Interpretation is independent of the particular computer being represented, analogous to heap-independent interpretations in [4]: $[\![E]\!]\,a \in Val$, where $dom\,(a)$ includes the free variables of E.

2.2 Basics and Syntax

We will make use of some familiar classical propositional connectives, some perhaps less-familiar temporal connectives, and a 'spatial' connective from a more recent logic. The familiar classical connectives are 'if, then', 'and', 'or', and 'not' and the familiar first-order quantifiers are 'there exists' and 'for all'.

Before marching on with definitions, we briefly describe the intention of the less common operators which we use. The operators 'until' and 'since' are both temporal, whose definition we take from [17]. 'Until' is about the future, and 'since' is about the past, but otherwise they are similar. We have 'ϕ until ψ' when the first formula ϕ is true now and into the future, for at least enough time such that the second formula becomes true at some time later. It is what one might expect when asking "Hold this cup until I get back." Though in our logic we will need to be explicit about the social assumption, in classical logic, of "If I return, then give me the cup." 'Since' is similar. We have 'ϕ since ψ' when at some point in the past ψ occurred, and ϕ has been occurring from then up through to the present. Again, as one might expect from “I have been sad since my cup broke.”

The final less-familiar connective we use is $*$ for ‘and, separately’. Usual, classical ‘and’ is collapsible. That is, “I have five coins and I have five coins” is, in classical logic, the same as “I have five coins.” The connective ‘and, separately’ is not collapsible. We take this connective from O’Hearn and Pym’s logic of bunched implications (BI) [8,19,23], a non-classical (‘substructural’) logic with a semantics that can be interpreted in terms of the composition and comparison of resources and which forms the basis for Separation Logic [13,24]. Separation Logic is a specific theory of BI for handling memory allocation—our direct starting point in this section. Readers may see [22] for an accessible introduction to Separation Logic.

Computers, like coins, are resources. We use Separation Logic because we want to be able to express “A computer is compromised and, separately, a computer is compromised” to be reasoned with as two computers are compromised, for example. The classical ‘and’ would lose this information that two computers are compromised, because the formula would collapse.

Following these intuitions, logical formulae are constructed inductively:

$$\begin{array}{rcc}
\phi, \psi ::= & \alpha & \text{Atomic formulae} \\
\mid & \bot & \text{Falsity} \\
\mid & \phi \Rightarrow \psi & \text{Material implication} \\
\mid & \texttt{emp} & \text{Empty content} \\
\mid & \exists x . \phi & \text{Existential quantification} \\
\mid & \phi \mathcal{U} \psi & \text{Temporal Until} \\
\mid & \phi \mathcal{S} \psi & \text{Temporal Since} \\
\mid & \phi * \psi & \text{Spatial conjunction}
\end{array}$$

Atomic formulae include equality and points-to relations, and predicates.

$$\begin{array}{rcc}
\alpha ::= & E = E' & \text{Equality} \\
\mid & E \mapsto E_1, E_2 & \text{Points to} \\
\mid & P\left(\left(Val_1, E_1\right), \left(Val_2, E_2\right)\right) & \text{Relational predicate} \\
\mid & \ldots &
\end{array}$$

In [13], points-to is defined as a three-place relation, $E \mapsto E_1, E_2$. [4] contains both a simple points-to relation, $E \mapsto E'$ and a higher-order concept of lists that treats the properties of lists as primary, rather than their contents. Our goal is not to analyze details of doubly-linked lists or higher-order lists. Our syntax does not treat lists directly. However, this three-place syntax provides a way to separate a large data element into arbitrary chunks while preserving their order. This works for memory, files on disk, and network packets. An example of why this is useful: we can represent malware analysis techniques, such as segment hashing, by representing properties of a connected series of expressions. However, our intention is not to be exhaustively faithful to the file-system representation. If the segments of a large file are not of interest, we may elide the details of the file system block size and the linked list that actually composes the file contents.

The usual classical and temporal symbols are defined from available formulae:

- negation; i.e., ‘not’, is $\neg\phi \overset{\text{def}}{=} \phi \Rightarrow \bot$
- truth is simply not false; i.e., $\top \overset{\text{def}}{=} \neg\bot$
- conjunction; i.e., ‘and’ is customarily $\phi \vee \psi \overset{\text{def}}{=} (\neg\phi) \Rightarrow \psi$
- disjunction; i.e., ‘or’ is thus $\phi \wedge \psi \overset{\text{def}}{=} \neg(\neg\phi \vee \neg\psi)$
- ‘for all’ is in terms of the existential, $\forall x.\phi \overset{\text{def}}{=} \neg\exists x.\neg\phi$
- ‘at least once in the future’ relates to until, $\Diamond\phi \overset{\text{def}}{=} \top\mathcal{U}\phi$
- ‘henceforth’ is $\Box\phi \overset{\text{def}}{=} \neg\Diamond\neg\phi$
- analogously, ‘at least once in the past’ is $\diamondminus\phi \overset{\text{def}}{=} \top\mathcal{S}\phi$
- and ‘has always been’ is $\boxminus\phi \overset{\text{def}}{=} \neg\diamondminus\neg\phi$.

We follow [15], in that we do not have a simple ‘next’ temporal operator. For various reasons [15] lays out, and we feel a choice that is validated by how incident analysts reason in our case studies, we primarily care about observable changes, not the precise sequence that brings those changes about.

2.3 Model

Our model is designed to support incident response reasoning by embedding the most important objects of analysis as the basis of the model. We keep the three salient types of computing resources separate, and index by time. Each resource is a partial monoid with composition operator and unit.

$(R_M, \cdot_M, e_M)$ for processor and RAM (M for memory)
$(R_D, \cdot_D, e_D)$ for file storage (D for disk)
$(R_N, \cdot_N, e_N)$ for network bandwidth (N for network)

where, for $i \in \{M, D, N\}$, R_i is a set of resource elements of the given type, $\cdot_i : R_i \times R_i \rightharpoonup R_i$ is a partial function operating on resources of the given type, and e_i is the unit element of $\cdot_i$ such that for all $r \in R_i$ it is the case that $r \cdot_i e_i = r = e_i \cdot_i r$.

More concretely, each R_M, R_D, R_N is composed of (address, content) pairs analogous to (stack, heap) pairs. We define $\mathbf{m}{::=}s, h$ for $\mathbf{m} \in R_M$, $\mathbf{d}{::=}\delta, \beta$ for $\mathbf{d} \in R_D$, and $\mathbf{n}{::=}\kappa, \upsilon$ for $\mathbf{n} \in R_N$. These sub-parts of the resources are proper subsets of the address and content defined above. The fact that $s \in \mathsf{S}$ with $\mathsf{S} \subset A$ and $h \in \mathsf{H}$ with $\mathsf{H} \subset C$ makes the usual stack-heap model of separation logic somehow contained in our address-content model. Further, we define $\delta \in \mathsf{N}$ for $\mathsf{N} \subset A$ and $\beta \in \mathsf{B}$ for $\mathsf{B} \subset C$ (for inodes and file blocks). For network host addresses and data units (i.e., packets), $\kappa \in \mathsf{K}$ for $\mathsf{K} \subset A$ and $\upsilon \in \mathsf{U}$ for $\mathsf{U} \subset C$.

Formally, these three resource monoids could be considered as one monoid $\mathbf{R} = (R, \cdot, \mathbf{E})$ where $R = R_M \uplus R_D \uplus R_N$ (the disjoint union of the resources), composition $\cdot$, $\cdot : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ such that

$$\cdot\,(r_1, r_2) ::= \begin{cases} r_1 \cdot_i r_2 & \text{if } r_1, r_2 \in R_i \\ \text{undefined} & \text{otherwise} \end{cases}$$

and $\mathbf{E} = \{e_M, e_D, e_N\}$

where $\mathbf{E} \cdot r ::= \begin{cases} \bigcup_{e \in \mathbf{E}} r \cdot_i e = \{r\} = \bigcup_{e \in \mathbf{E}} e \cdot_i r & \text{if } r \in R_i \\ \text{undefined} & \text{otherwise} \end{cases}$

The definitions of $\cdot$ and a set of units are adapted from [3, Definition 2.3]. These definitions will be used to describe the state of a computer or a computer network at a given time as a composition of different programs, files, and network activity.

Incident analysis needs a notion of time and changes. Therefore, we adopt a linear time model composed of a sequence of states. Each state is represented by an element $r \in \mathbf{R}$. We define a history $H \in \mathcal{H}$ as a ordered finite set

$$H ::= \left\{r^1, r^2, \ldots, r^t, \ldots, r^T\right\},$$

with $T \in \mathbb{N}$. (H, t) uniquely identifies the state $r^t \in \mathbf{R}$. The length of a history is $|H| = T$. There is no notion of absolute time or a "wall clock." The time T indicates a sequence without any claims about the time between transitions.

History Monoid. We define a monoid, $\mathbf{H} = (\mathcal{H}(\mathbf{R}), \circ, e)$ where $\mathcal{H}$ is the set of histories H (defined above) that can be constructed using a given resource monoid $\mathbf{R}$; $\circ : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$; unit e to be the empty history with $|e| = 0$. More specifically, we define $\circ$ as:

$$(H_1 \circ H_2, t) ::= \begin{cases} (r_1^t \cdot r_2^t, t) \text{ for } r_1^t \in H_1 \text{ and } r_2^t \in H_2 & \text{if } |H_1| = |H_2| \\ (H_2, t) & \text{if } H_1 \prec H_2 \\ (H_1, t) & \text{if } H_2 \prec H_1 \\ \text{undefined} & \textit{otherwise} \end{cases}$$

Here $H_1 \prec H_2$ indicates that one history is contained in the other. We define four conditions that must all be met for this to hold. Specifically, $H_1 \prec H_2$ iff

1. $|H_1| < |H_2|$, where $|H_1| = T$, $|H_2| = T'$; and
2. for all $r_1^t \in H_1$, with $t \in T$, there exists some $r_2^{t'} \in H_2$ with $t' \in T'$ such that $r_1^t = r_2^{t'}$ and $t \leq t'$; and
3. for all $r_2^{t'} \in H_2$ and given any r_1^t, r_1^x in H_1 with $t, x \in T$, it is the case that $r_2^{t'} = r_1^t$ and $r_2^{t'} = r_1^x$ iff $t = x$; and
4. for all r_1^t, r_1^x in H_1 with $t, x \in T$ such that $t < x$, it is the case that, for $r_2^{t'}, r_2^{x'} \in H_2$ with $t', x' \in T'$, we have $r_1^t = r_2^{t'}$ and $r_1^x = r_2^{x'}$ iff $t' < x'$.

The intuition for these requirements as expressing the concept of "contained in" is as follows. A smaller history is contained in a larger one. All the events of the smaller history appear in the larger one, in the same relative ordering. The only change permitted is that new events are inserted into the larger history; such inserted events can be interleaved in any way.

The unit e as the empty history behaves as expected.

$$H \circ e = H = e \circ H$$

Proof of identity by cases. We have$|e| = 0$, so either

1. $|H| = 0$, that is H is e, thus we have to prove $e \circ e = e$
   (a) This is true. We follow $r^t ::= r_1^t \cdot r_2^t$. However, $T = 0$ so there are no elements to compose. The result is the history of length 0, namely, e.
2. $|H| \geq 1$
   (a) Requirement 1 for $\prec$ holds $(0 < 1)$.
   (b) Requirement 2 holds vacuously (all $r_1^t \in e$ is $\emptyset$).
   (c) Requirement 3 holds vacuously, without r_1^t, r_1^x to compare.
   (d) Requirement 4 holds vacuously, without r_1^t, r_1^x to compare.

One might think the unit for $\circ$ could be the history containing just the unit element $\mathbf{E}$ (recall $\mathbf{E} = \{e_M, e_D, e_N\}$). However, if defined thus, requirement 2 for $\prec$ might fail if there is no element of H in $H \circ e$ such that $(H, t) = \mathbf{E}$. Then $H \circ e$ could be undefined for $|H| > 1$, in which case $H \circ e = H = e \circ H$ would not hold as required. Every history could start with the unit element to make this true by construction, but that seems unnatural. Therefore the unit of $\circ$ should be the empty history $|e| = 0$.

A history will be used to represent a hypothesis for the series of events and changes to the resources of a computer system during the course of the incident. Combining histories can represent, for example, combining explanations of simultaneous events on two different locations on the network.

2.4 Semantics

The semantics of the atomic expressions are many-sorted operations. To unfold the truth value of an expression, recall $(H, t) \overset{\text{def}}{=} [(s^t, h^t), (\delta^t, \beta^t), (\kappa^t, \upsilon^t)]$.

$$[(s^t, h^t), (\delta^t, \beta^t), (\kappa^t, \upsilon^t)] \models E = E' \text{ iff } \begin{cases} [\![E]\!]\, s^t = [\![E']\!]\, s^t \\ [\![E]\!]\, \delta^t = [\![E']\!]\, \delta^t \\ [\![E]\!]\, \kappa^t = [\![E']\!]\, \kappa^t \end{cases}$$

We can abbreviate this as

$$H, t \models E = E' \text{ iff } [\![E]\!]\, a^t = [\![E']\!]\, a^t$$

Because these three resources are disjoint (namely $\mathsf{S} \subset A; \mathsf{N} \subset A; \mathsf{K} \subset A$ and $\mathsf{S} \cap \mathsf{N} = \mathsf{S} \cap \mathsf{K} = \mathsf{N} \cap \mathsf{K} = \emptyset$), only one of the three interpretations will be valid. Namely, only one of $[\![E]\!]\, s$ or $[\![E]\!]\, \delta$ or $[\![E]\!]\, \kappa$ can hold for any E, or they are equivalent. Only one exists because for $[\![E]\!]\, a$ to be interpretable, $dom\,(a)$ must include the free variables of E. The domains of s, δ, κ are disjoint by definition. If there are no free variables in E, then $[\![E]\!]\, s = [\![E]\!]\, \delta = [\![E]\!]\, \kappa$.

Similarly, points-to can be defined over the three disjoint parts of the model at a given time, and then abbreviated in terms of elements of A and C:

$$\begin{aligned} &[(s^t, h^t), (\delta^t, \beta^t), (\kappa^t, \upsilon^t)] \models E \mapsto E_1, E_2 \\ &\text{iff} \begin{cases} h^t\,([\![E]\!]\, s^t) = \langle [\![E_1]\!]\, s^t, [\![E_2]\!]\, s^t \rangle & \{[\![E]\!]\, s^t\} = dom\,(h^t) \\ \beta^t\,([\![E]\!]\, \delta^t) = \langle [\![E_1]\!]\, \delta^t, [\![E_2]\!]\, \delta^t \rangle & \{[\![E]\!]\, \delta^t\} = dom\,(\beta^t) \\ \upsilon^t\,([\![E]\!]\, \kappa^t) = \langle [\![E_1]\!]\, \kappa^t, [\![E_2]\!]\, \kappa^t \rangle & \{[\![E]\!]\, \kappa^t\} = dom\,(\upsilon^t) \end{cases} \end{aligned}$$

which we abbreviate as

$$H, t \models E \mapsto E_1, E_2 \text{iff} \left\{ [\![E]\!]\, a^t \right\} = dom\left(c^t\right) \text{ and } c^t\left([\![E]\!]\, a^t\right) = \left\langle [\![E_1]\!]\, a^t, [\![E_2]\!]\, a^t \right\rangle$$

The element `emp` actually represents a set of three related elements: $\left\{ \overset{M}{\texttt{emp}}, \overset{D}{\texttt{emp}}, \overset{N}{\texttt{emp}} \right\}$. The semantics for `emp` is defined as

$$\begin{aligned}
\left[\left(s^t, h^t\right), \left(\delta^t, \beta^t\right), \left(\kappa^t, \upsilon^t\right)\right] &\models \overset{M}{\texttt{emp}} \text{ iff } h^t = [] \\
\left[\left(s^t, h^t\right), \left(\delta^t, \beta^t\right), \left(\kappa^t, \upsilon^t\right)\right] &\models \overset{D}{\texttt{emp}} \text{ iff } \beta^t = [] \\
\left[\left(s^t, h^t\right), \left(\delta^t, \beta^t\right), \left(\kappa^t, \upsilon^t\right)\right] &\models \overset{N}{\texttt{emp}} \text{ iff } \upsilon^t = [] \\
H, t &\models \texttt{emp} \text{ iff } \overset{M}{\texttt{emp}} \text{ and } \overset{D}{\texttt{emp}} \text{ and } \overset{N}{\texttt{emp}}
\end{aligned}$$

Here, $h^t = []$, $\beta^t = []$, and $\upsilon^t = []$, represent the empty heap, empty file system, and empty network, respectively.

The semantics for a relational predicate, P, is given by

$$H, t \models P\left(\left(Val_1, E_1\right), \left(Val_2, E_2\right)\right) \text{ iff } (H, t) \in \mathbb{V}\left[P\left(\left(Val_1, E_1\right), \left(Val_2, E_2\right)\right)\right]$$

Here $\mathbb{V} : \mathbb{A} \to \mathcal{P}(States)$ is the valuation function from the set $\mathbb{A}$ of atoms of $P\left(\left(Val_1, E_1\right), \left(Val_2, E_2\right)\right)$ to the powerset of possible states of the form (H, t).

The other semantic clauses are as follows:

$$\begin{array}{lll}
H, t \models \phi \Rightarrow \psi & \text{iff} & \text{if } H, t \models \phi \text{ then } H, t \models \psi \\
H, t \models \exists x.\phi & \text{iff} & \text{for some } v \in Val.\, [a|x \mapsto v], c \models \phi \\
H, t \models \phi \mathcal{U} \psi & \text{iff} & \text{for some } i \in T \text{ with } i \geq t \text{ and } (H, i) \models \psi \text{ such that} \\
 & & \text{for all } j \in T \text{ with } t \leq j < i \text{ it is the case } (H, j) \models \phi \\
H, t \models \phi \mathcal{S} \psi & \text{iff} & \text{for some } i \in T \text{ with } i \leq t \text{ and } (H, i) \models \psi \text{ such that} \\
 & & \text{for all } j \in T \text{ with } i < j \leq t \text{ it is the case } (H, j) \models \phi \\
H, t \models \phi * \psi & \text{iff} & \text{for some } H_1, H_2 \text{ such that } H_1 \# H_2 \text{ and } H_1 \circ H_2 = H \\
 & & \text{where } H_1, t \models \phi \text{ and } H_2, t \models \psi
\end{array}$$

Here $H_1 \# H_2$ indicates the histories are pointwise disjoint. $H_1 \# H_2$ is true if and only if the following conditions hold:

1. $|H_1| = |H_2| = T$; and
2. For all $\left[\left(s_1^t, h_1^t\right), \left(\delta_1^t, \beta_1^t\right), \left(\kappa_1^t, \upsilon_1^t\right)\right] \in H_1$ and $\left[\left(s_2^t, h_2^t\right), \left(\delta_2^t, \beta_2^t\right), \left(\kappa_2^t, \upsilon_2^t\right)\right] \in H_2$ it is the case that, for all $t \in T$:
   (a) $dom\left(h_1^t\right) \cap dom\left(h_2^t\right) = \emptyset$ and
   (b) $dom\left(\beta_1^t\right) \cap dom\left(\beta_2^t\right) = \emptyset$ and
   (c) $dom\left(\upsilon_1^t\right) \cap dom\left(\upsilon_2^t\right) = \emptyset$.

2.5 Abduction

These tools will allow us to capture abduction.Abduction would naturally be grouped into a trio with deduction and induction. These terms have long, problematic histories of usage. Deduction requires a proof theory, and because one can justifiably define different proof theories for different purposes [21], 'deduction' is not just one thing. But generally 'deduction' captures the reasoning from premises to conclusions following explicit rules. We discuss proof theory briefly in Sect. 2.6. 'Induction' has received voluminous attention, since Hume in the 1740 s [9]. It roughly means concluding that because something has been the case before, it will be again. A more fruitful discussion might be had under the topic of how we generalize from what we know. Generalization methods will generate the heuristics we need for the logic. However, we leave generalization aside for now; there are other discussions of effective methods (see, e.g., [29]).

Abduction is neither deduction nor induction. Abduction is the generation of an explanation, which can then be evaluated against available evidence [2, CP 5.171]. More formally, abduction asks what (minimal) formula needs to be added to a proposition such that it will be satisfied. As [4] demonstrates, abduction is automatable as long as the problem space is constrained, checking the validity of hypothetical additions is scalable, and human heuristics for generating additions can be encoded in the logic. Attack ontologies will serve as these heuristics for incident analysis. We will endeavor to represent one common attack ontology—the intrusion kill chain [12]—in our logic. We will also demonstrate that we can link existing knowledge bases, such as Snort rules, into this structure. Therefore, we are confident a system could be built that instrumented a computer network, ingested security-relevant information, and, given a security incident, used our logic to assist in the process of abducing explanations of how an adversary penetrated the network. Given this decision support, we would then imagine testing and improving different abduction rules in a scientific manner.

2.6 On the Metatheory of the Security Incident Analysis Logic

Generally, when setting up and explaining a system of logic, one gives a language (of propositions) and a semantics specified by a class of models together with a satisfaction relation which specifies which propositions are true in which parts of an arbitrary model. Typically, one also gives a proof system—that is, a collection of inference rules—which determines which propositions are provable. The first meta-theoretic challenge is then to establish that the provable things are also true in the models (soundness) and that there is model for which the notion of truth specified in the semantics coincides with the notion of provability specified by the inference rules (completeness). This, together with other metatheoretic analyses, is what assures us that a logic makes good sense.

In this section, we have described a logic for analysing security incidents. We have defined the logic by giving its propositional language together with a semantics given by a specific model together with a satisfaction relation which determines which propositions are true in which parts of the given model.

So, given that we haven't done all the usual work, why are we confident that the logic is a good one? Although the logic we have defined may look quite exotic, it is, in fact, based on a combination of some quite well-understood constructions together with a specific concrete model. In this respect, its definition

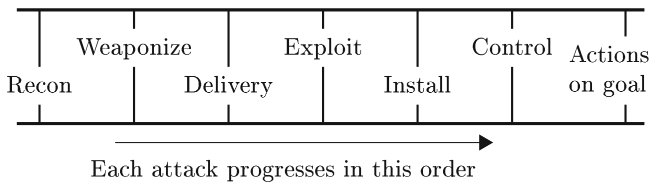

Fig. 1. The intrusion kill chain, as explained by [12]. We will add more detail to this attack ontology by specifying certain aspects in our incident analysis logic.

somewhat resembles that of the logic from which it draws much inspiration, namely Separation Logic [4,23].

In short, for general mathematical reasons about how logics are constructed, we can be confident that the logic will work properly in the established senses.

3 A Worked Example

The "kill chain" was introduced by Lockheed Martin to explain an abstract pattern they observed in attacks targeting their organization [12]. It is a useful model of computer network attacks, because it helps inform the incident analyst about expected sorts of activities, against what sorts of entities, and their organization. The abstract nature of the kill chain makes it a good example to be expressed in our logic. It also models a useful unit of incident analysis – a single attack. Multiple attacks are almost always sequenced to achieve an adversary's overall goal during a campaign. Also, most attacks do not succeed, so usually many attacks occur in parallel. Therefore, modelling a single attack should be a fruitful example because we can compositionally build on it to analyze security incidents.

Figure 1 summarizes the steps in the kill chain. The mechanistic expression of the kill chain elaborated in [27] also guides the expression in our logic.

Our example is to turn this conceptual model of the kill chain into a set of logical statements of pre- and post-conditions that express useful abduction heuristics. However, we need to realign the components of the model. Our logic talks about observable computer activity, and as such the humans implicit in the kill chain have no place in our logic. Their interests are represented in the definition of our predicates. For example, the truth values of *compromised*() will depend on the security policy of the defender.

What counts as malware or an exploit is also dependent on the point of view of the agents. In our logic, we model only software instructions, computer systems, and bit-strings. These categories are intention-neutral. Malware is a subclass of software. Strictly, we do not discuss software (as this implies a complete product), but rather just instruction sets—a *computation*. But we shall not dictate how malware is classified as such. One benefit of our logic is to express precisely how an analyst determines how to differentiate malware from benign computations. Descriptions of what behaviors are indicative of malicious versions of those elements will be contingent.

To define our representation of a computation (i.e., software, functions, etc.), we adapt Hoare-Floyd logic. Hoare logic is a mainstay of program verification. It is primarily concerned with statements of the form $\{\phi\}\,\mathcal{C}\,\{\psi\}$, where ϕ is pre-conditions, ψ is post-conditions, and $\mathcal{C}$ is some specific computation. The goal of Hoare logic is to verify that ψ can be guaranteed to be satisfied if $\mathcal{C}$ executes in an environment that satisfies ϕ.

The construction of Hoare logic is about the details of $\mathcal{C}$ and whether we can demonstrate post-conditions given pre-conditions. We are going to turn this on its head. The incident responder knows a post-condition, usually some security violation, and wants to understand more about the pre-conditions and software.

The computation $\mathcal{C}$ can be described in various levels of detail. This is an important benefit. Our logic, so defined, permits description of programmatic details. Malware reverse engineering tries to construct details of an unknown $\mathcal{C}$. Incident analysis is primarily involved in a higher level task, merely constraining the observable traces in the system, not how some $\mathcal{C}$ made these changes. Therefore, while knowing malware details is helpful, because it narrows the potential pre- and post-conditions, we leave discussion of how $\mathcal{C}$ works in malware for future work. Practicing incident responders should reduce attention regards malicious logic as simply the Hoare triple $\{\phi\}\,\mathcal{C}\,\{\psi\}$ where ϕ and ψ are known. This approach to knowledge is essentially the programming principle of encapsulation. If we know what goes in and what comes out, we do not need to know how it works to reason about impacts on our system.[1]

We represent a computer system as σ, taken from the systems known to the analyst. The full complement of systems is represented by $\mathbf{S}$. At a given time t, the system is σ^t. The system σ^t is shorthand for a cluster of resources $[(s^t, h^t), (\delta^t, \beta^t), (\kappa^t, \upsilon^t)]$. Therefore, at any given time t, the state of the world (H, t) might be decomposed into one or more systems $\sigma_1^t \cdot \sigma_2^t \cdot \ldots \cdot \sigma_n^t$. The concept of system is therefore merely a shorthand for a cluster of resources that the incident analyst is interested to treat as a unit of analysis.

Our third and final entity, bit-strings are a type of expression E. Usually we represent strings in human-readable form. Human-readable strings can be represented as integers, so the syntax for E remains unchanged. We elide the details of local encodings (ASCII vs. unicode vs. hexadecimal, big- vs. little-endian, etc.) that complicate mapping between strings and integers. Notating strings as strings instead of expressions is merely a syntactic convenience.

Given computations and systems, we can define all the predicates we need:

- $\mathit{compromised}\,(\sigma^t)$
- $\mathit{hostile}\,(\sigma^t)$
- $\mathit{malicious}\,(\mathcal{C})$
- $\mathit{trusts}\left(\sigma_1^t, \sigma_2^{t'}\right)$ (often with $t = t'$)
- $\mathit{match}\,(\mathit{string}_1, \mathit{string}_2)$
- $\mathit{vulnerable}\,(\sigma^t, \mathcal{C})$
- $\mathit{exploited}\,(\sigma^t, \mathcal{C})$

[1] Any given $\{\phi\}\,\mathcal{C}\,\{\psi\}$ for a program will be treated as a hypothesis, and one that given sufficient evidence might be overturned and modified.

Compromised, hostile, and malicious have the intuitive meanings. In our current set of definitions, these have binary truth values. We recognize analysts may be interested in intermediate values; however, we leave an extension of the logical definitions to a many-valued logic as future work.

Note that our intention here is that the compromised system is internal, under defender ownership, whereas a hostile system is on the Internet, not owned by the defender. Therefore, a different reasonable definition would be to define an ownership predicate, and define $\mathit{compromised}\left(\right)$ in terms of hostile and owned. That is, there are multiple compatible ways to represent relevant concepts. We select the above as a viable definition, not the only one.

The predicate $\mathit{trusts}\left(\sigma_1^t, \sigma_2^{t'}\right)$ is a relationship between two systems. Although it is an oversimplification, for the time being we reduce trust to the ability to communicate. More specifically, receive information. That is, given an address $a_1 \in A$ such that $a_1 \subset \sigma_1^t$ and address $a_2 \in A$ such that $a_2 \subset \sigma_2^t$ and any expression E, we have $\mathit{trusts}\left(\sigma_1^t, \sigma_2^{t'}\right)$ just in case that $(a_1 \mapsto E) \Rightarrow \Diamond (a_2 \mapsto E)$.

This is an abstract concept of communication. It just says that if some address in system one points to an expression, somehow eventually an address in system two comes to point to the same expression. The reason this is trust, and not chance, is that this relationship holds for any expression. This definition abstracts away from how that communication is executed. A real security policy may restrict which expressions are permitted or disallowed. We leave such definitions of a trust predicate as future work.

The predicate $\mathit{match}\left(\right)$ represents a common use case in incident analysis and computer network defense: pattern matching. Tools such as intrusion detection systems, firewall ACLs, and spam email detection all rely on matching incoming communications to known patterns of interest. These patterns are signatures or blacklists of malicious behavior.

We define the semantics of $\mathit{match}\left(\mathit{string}_1, \mathit{string}_2\right)$ such that:

$$\left[\left(s^t, h^t\right), \left(\delta^t, \beta^t\right), \left(\kappa^t, \upsilon^t\right)\right] \models \mathit{match}\left(\mathit{string}_1, \mathit{string}_2\right)$$

just in case

$$\mathit{in}\left(\left[\left(s^t, h^t\right), \left(\delta^t, \beta^t\right), \left(\kappa^t, \upsilon^t\right)\right], \mathit{string}_2\right) \wedge \mathit{string}_1 = \mathit{string}_2$$

The $\mathit{in}\left(\right)$ predicate holds just in case

$$\begin{aligned}
&[\![\mathit{string}_2]\!] \in \mathit{dom}\left(s^t\right) \vee [\![\mathit{string}_2]\!] \in \mathit{dom}\left(h^t\right) \vee \\
&[\![\mathit{string}_2]\!] \in \mathit{dom}\left(\delta^t\right) \vee [\![\mathit{string}_2]\!] \in \mathit{dom}\left(\beta^t\right) \vee \\
&[\![\mathit{string}_2]\!] \in \mathit{dom}\left(\kappa^t\right) \vee [\![\mathit{string}_2]\!] \in \mathit{dom}\left(\nu^t\right)
\end{aligned}$$

We may abbreviate this as $\mathit{in}\left(\sigma^t, \mathit{string}\right)$ or $\mathit{in}\left(\left(H, t\right), \mathit{string}\right)$. If we wish to emphasize a certain type of string only occurs in the contents of files, for example, we may elide the other variables and write $\mathit{in}\left(\beta^t, \mathit{string}\right)$.

The equality operator is expression equality as defined in Sect. 2.4, since strings are expressions. Specifically, if strings are understood as integers, the expressions will have no free variables and so it becomes the usual integer equality.

We write $\sigma^t \models \{\phi\}\,\mathcal{C}\,\{\psi\}$ just in the case that there is some content $c \in C$ and $\sigma^t \models \mathit{match}\,(\mathcal{C}, c) \Rightarrow (\phi \Rightarrow \Diamond\psi)$. This assumes that the computation $\mathcal{C}$ terminates. But we are primarily concerned with malware that has successfully run, so this should not cause great trouble. Furthermore, we have defined time as finite, so termination can always be defined as the state at (H, t) when $t = T$.

We then propose to define $\mathit{vulnerable}\,(\sigma^t, \mathcal{C})$ to hold iff $\sigma^t \models (\{\phi\}\,\mathcal{C}\,\{\psi\}) \wedge \phi \wedge \mathit{malicious}\,(\mathcal{C})$. The real-world impact if $\mathit{vulnerable}\,(\sigma^t, \mathcal{C})$ holds is a bad security situation. Such a system can be exploited at the will of the adversary.

To differentiate from the less severe situation where a system is vulnerable but exploit code is not present, we define $\sigma^t \models \mathit{vul}\,(\phi)$. This is a syntactic convenience; it means only that $\sigma^t \models \phi$ and that ϕ is the precondition for the execution of some malware.

Vulnerability is not the same as exploitation (in the traditional terminology of computer security). Exploit also requires access, which we can define in terms of trusting, execution, etc. However, simply the state of having been exploited, $\mathit{exploited}\,(\sigma^t, \mathcal{C})$, we can define as $\sigma^t \models \diamondminus(\{\phi\}\,\mathcal{C}\,\{\psi\}) \wedge \mathit{malicious}\,(\mathcal{C}) \wedge \psi$.

3.1 A Logic of the Kill Chain

The kill chain provides the incident analyst with abduction heuristics for abducing the pre-conditions that lead to observed post-conditions. Thus, we can define pre- and post-conditions that we expect from each of the seven steps of the kill chain. If we observe the post-conditions of one, we can abduce its pre-conditions. We will use the kill chain to provide the basic structure of a single attack. Once this is complete, we will suggest how the logic can group attacks together into campaigns. Thirdly, we can specify more specific conditions for kill chain steps at a level of detail that is compatible with tools available to practicing incident analysts.

The last step in the kill chain is the first that an incident analyst is likely to observe. Thus our measure of time starts with $t = T$, the end of the history, and works backwards to $t = 0$. Because we have no absolute notion of time, each discrete phase moves back time one step. In this way, we will continue to step backwards through the attack from the end to the beginning:

– Action on Objectives, the final state: the system is under adversary control
  • *Post-condition* (observed): $H, t \models \mathit{Compromised}\,(\sigma_1^t)$ for $t = T$.
  • *Pre-condition*: C&C, defined as: there is some σ_2 such that $H, t \models \mathit{trusts}\,(\sigma_1^t, \sigma_2^t) \wedge \mathit{hostile}\,(\sigma_2^t)$ for $t = T - 1$.

This does not tell the analyst much, but it importantly identifies that there must be some hostile system that the defender's system has come to trust. Unwinding the next steps backwards would shed light on how.

- Command and control
  - *Post-condition* (observed): C&C, as defined above
  - *Pre-condition*: Installation of malware, that is $\sigma_1^t \models \Box\left(\left\{\widehat{\phi_{\mathcal{C}_1}}\right\}\mathcal{C}_1\left\{\widehat{\psi_{\mathcal{C}_1}}\right\}\wedge\widehat{\phi_{\mathcal{C}_1}}\right)$, for $t = T-2$. The $\Box$ indicates that the malware will be able to execute indefinitely into the future, not just once.

Where $\left\{\widehat{\phi_{\mathcal{C}_1}}\right\}\mathcal{C}_1\left\{\widehat{\psi_{\mathcal{C}_1}}\right\}$ as follows:

$\widehat{\psi_{\mathcal{C}_1}}$ is a post-condition for the adversary's objectives, namely at minimum establishing a communication channel; i.e., $H, t \models \textit{trusts}\,(\sigma_1^t, \sigma_2^t)$ for $t = T-1$. Discovery of further unobserved objectives is likely one investigative goal.

$\widehat{\phi_{\mathcal{C}_1}}$ is the pre-conditions for the malware to run. These may simply be the post-conditions of the installer (i.e., $\widehat{\psi_{\mathcal{C}_2}}$, defined below), but may include what type of system the adversary can or wants to target.

A more flexible definition of the pre-conditions for command and control would be $\left(\left\{\widehat{\phi_{\mathcal{C}_1}}\right\}\mathcal{C}_1\left\{\widehat{\psi_{\mathcal{C}_1}}\right\}\right)\mathcal{U}\phi$, for some ϕ, instead of $\Box\left(\left\{\widehat{\phi_{\mathcal{C}_1}}\right\}\mathcal{C}_1\left\{\widehat{\psi_{\mathcal{C}_1}}\right\}\right)$.

- Installation of $\mathcal{C}_1$ (the main malware) by $\mathcal{C}_2$ (a downloader, installer, etc.)
  - *Post-condition* (observed): Installation, captured by $\sigma_1^t \models \left\{\widehat{\phi_{\mathcal{C}_1}}\right\}\mathcal{C}_1\left\{\widehat{\psi_{\mathcal{C}_1}}\right\}\wedge\widehat{\phi_{\mathcal{C}_1}}$, for $t = T-2$.
  - *Pre-condition*: Exploitation; i.e., $\sigma_1^t \models \textit{exploited}\,(\sigma_1^t, \mathcal{C}_2)$, for $t = T-3$.

Note that the installation post-condition is weaker than the command and control pre-condition. The post-condition is what can be observed, but the pre-condition is abduced. In this context, the analyst should not assume the malware will stop, but rather that it will continue running indefinitely. Of course, like all abductions, this hypothesis might be changed by further observations.

Here $\left\{\widehat{\phi_{\mathcal{C}_2}}\right\}\mathcal{C}_2\left\{\widehat{\psi_{\mathcal{C}_2}}\right\}$ is as follows.

$\widehat{\psi_{\mathcal{C}_2}}$ contains at least that $\sigma_1^t \models \left(\left\{\widehat{\phi_{\mathcal{C}_1}}\right\}\mathcal{C}_1\left\{\widehat{\psi_{\mathcal{C}_1}}\right\}\right)\wedge\widehat{\phi_{\mathcal{C}_1}}$, for $t = T-2$. I.e., system one both stores the malware and is configured such that it can run.

$\widehat{\phi_{\mathcal{C}_2}}$ is a pre-condition containing at least the transfer of data necessary for the installation; i.e., there is some σ_3 such that $H, t \models \textit{trusts}\,(\sigma_1^t, \sigma_3^t)$, for $t = T-4$.

- Exploitation of system σ_1 by an exploit $\mathcal{C}_3$.
  - *Post-condition* (observed): $\sigma_1^t \models \left\{\widehat{\phi_{\mathcal{C}_2}}\right\}\mathcal{C}_2\left\{\widehat{\psi_{\mathcal{C}_2}}\right\}\wedge\widehat{\psi_{\mathcal{C}_2}}$, for $t = T-3$.
  - *Pre-condition*: $\sigma_1^t \models \textit{vulnerable}\,(\sigma_1^t, \mathcal{C}_3)$, for $t = T-5$.

Here $\left\{\widehat{\phi_{\mathcal{C}_3}}\right\}\mathcal{C}_3\left\{\widehat{\psi_{\mathcal{C}_3}}\right\}$ is as follows:

$\widehat{\psi_{\mathcal{C}_3}}$ contains at least $\sigma_1^t \models \left(\left\{\widehat{\phi_{\mathcal{C}_2}}\right\}\mathcal{C}_2\left\{\widehat{\psi_{\mathcal{C}_2}}\right\}\right)\wedge\widehat{\phi_{\mathcal{C}_2}}$, for $t = T-4$. We say "at least" here because the exploit may or may not delete itself, for example, so in general additional traces on the system cannot be specified.

$\widehat{\phi_{\mathcal{C}_3}}$ represents the exploited vulnerability and any targeting by the adversary.

- Delivery of an exploit
  - *Post-condition* (observed): There exists content $c, c' \in C$ such that it is the case $\sigma_1^t \models match\,(\mathcal{C}_2, c) * match\,(\mathcal{C}_3, c')$, for $t = T - 6$.
  - *Pre-condition*: There is σ_4 such that $(H, t) \models trusts\,(\sigma_1^t, \sigma_4^t)$, for $t = T - 7$.

The delivery phase does not assume the exploit runs, just that it reaches the defender's system from somewhere. We abduced the existence that system, σ_4.

- Weaponization against an observed vulnerability
  - *Post-condition* (explicitly unobserved): This is the creation of the malware. It also might include all the work the adversary did to discover the vulnerability, etc.
  - *Pre-condition*: The reconnaissance was successful and the adversary learns that the system $\sigma_1^t \models vul\,(\phi)$ for some ϕ, for $t = T - 8$.

Weaponization is an abduced step. Because it occurs local to the adversary, the defender almost never observes it, but knows that it must happen.

- Reconnaissance on target systems
  - *Post-condition*: Observable communication between σ_5 and σ_1. That is, $(H, t) \models trusts\,(\sigma_1^t, \sigma_5^t) \wedge \psi$, for $t = T - 9$, where ψ represents the information communicated. In some situations, it may be possible to learn what vulnerability is likely communicated, that is $\psi \Rightarrow vul\,(\phi)$.
  - *Pre-condition*: There exists σ_5 such that $(H, t) \models trusts\,(\sigma_1^t, \sigma_5^t)$, for $t = 0$. Depending on the communication, it may be possible to put constraints on what cluster of resources represent σ_5.

The adversary-controlled systems $\sigma_5, \sigma_4, \sigma_3, \sigma_2$ may or may not be the same real-world system, sharing some combination of resources.

3.2 Composition of Attacks into a Campaign

To model a campaign of many attacks, we would join attacks together by $*$. This is particularly important because the compromised system σ_1 might be used to conduct further attacks locally. The postconditions of one attack might be preconditions for other attacks. It's important that this is $*$ and not $\wedge$, to count compromised machines and attacks as individuals.

A logical description of botnet operations, such as Zeus, should be possible by composing aspects and instances of the kill chain. Indeed, [5] accomplish something similar by stitching together kill chain instances with Bayesian belief statements. Incorporating existing tools such as MulVal [20], which helps with vulnerability management by logical discovery of impactful attack graphs, are promising areas for synergy with the logic presented here. We leave a worked campaign example for future work.

3.3 Using More Granular Knowledge

[27] use the kill chain as an example of mechanistic explanation and demonstrate incorporating a lower (mechanistic) level of explanation via a type of exploitation: drive-by download. In our logic, we can similarly refine our expressions. For example, known exploits would put constraints on $\left\{\widehat{\phi_{\mathcal{C}_3}}\right\} \mathcal{C}_3 \left\{\widehat{\psi_{\mathcal{C}_3}}\right\}$. We will demonstrate using a simpler example than drive-by downloads.

Integrating specific rules should enable automating the process of finding likely explanations. The incident analyst might have many thousands of potential specifications of various phases of the kill chain, derived from anti-virus signatures, blacklists, and so on. The logic mechanizes the inspection of which details are more likely to be at play in a given incident based on observations.

We will demonstrate how existing knowledge bases can be leveraged in this way via a Snort rule. An intrusion detection system (IDS) rule, such as Snort rules, is a structured statement of suspicious network activity. We consider an old, but representative, example rule from [25], which introduced Snort. Translations from anti-virus rules, etc., should be similarly easy.

```
alert tcp any any -> 192.168.1.0/24 143 (content:"|E8C0 FFFF
FF|/bin/sh"; msg:"IMAP Buffer Overflow detected!";)
```

This rule is a specification of the kill chain "Delivery" phase. Some parts are responses, such as "alert", which need not be represented in the logic. Similarly, annotation such as the "msg" field is useful, but would be implemented elsewhere.

This leaves essentially two elements of the rule. The header, which specifies the matching rules for packet headers, and the payload detection options (here, "content"). In this case, these aspects map to statements about the network, namely κ, ν. Specifically, header rules are about κ and content rules are about ν. This makes translation of such Snort rules relatively straightforward.

The network headers are simply communication between some external system, σ_4, and the defender's system σ_1. System σ_4 remains unconstrained, represented by `any any` for IP and port matches. However, we have two constraints on σ_1. Firstly, the system is 255 IP addresses, namely `192.168.1.0/24`. We represent this as the claim that there exists some $\kappa \in A$ such that

$$\sigma_1^t \models \mathit{dip}\,(192.168.1.0, \kappa) \vee ... \vee \mathit{dip}\,(192.168.1.255, \kappa) \wedge \mathit{dport}\,(143, \kappa)$$

The predicates $\mathit{dip}\,()$ and $\mathit{dport}\,()$ use the $\mathit{match}\,()$ predicate, defined to match specific parts of an IP packet header (destination IP and port, respectively).

The content is a string-matching constraint on the communication between σ_4 and σ_1. We change the notation for hexadecimal content from $|FF|$, as Snort uses, to $\underline{\texttt{FF}}$. Then this half of the Snort rule is easily translated; we assert there exists some $\nu \in C$ such that

$$\sigma_1^t \models \mathit{match}\,(\underline{\texttt{E8C0FFFFFF}}\texttt{/bin/sh}, \nu)\,.$$

This matches with an exploit, represented as $\mathcal{C}_3$ in our formulation. The Snort rule is the conjunction of these two statements.

Recall the broad statement of delivery in the kill chain. Transfer of data, including $\mathcal{C}_3$, from some σ_4 to σ_1. We have demonstrated how one can specify greater detail of these aspects. Specifying the specifics of all such attacks is a huge undertaking. For that reason, we have chosen an example – Snort rules – where much of this undertaking has already been collected and curated in machine-readable form. Such existing data bases of attack patterns should be readily leveraged by our incident analysis logic.

We should also propagate specifics forward in the kill chain. This example finds an attack against email servers. Therefore, we know more accurate preconditions for $\mathcal{C}_3$. Particularly, whether $\mathit{vulnerable}(\sigma_1, \mathcal{C}_3)$ holds. If σ_1 is not an email server, then it is not vulnerable. This sort of reasoning should allow the analyst to reduce the number of systems that need to be investigated as to whether the exploitation step was successful, for example.

4 Conclusions

One ambition for this logic is to represent the reasoning in [30]. This task requires a large – but finite – collection of observations, reasoning heuristics, hypothetical explanations, and deduced conclusions. We have not laid out these usage patterns in detail, but we are confident our tools would work similarly to Separation Logic, which has these features [4]. But the question may remain: why?

We envision three primary benefits to incident analysis (and perhaps cybersecurity broadly) from engaging with logical tools; namely, *communication*, *clarification*, and *decision-support potential*.

A logic such as the one we have sketched aids communication between analysts. In general, logical tools aid communication by reducing ambiguity. If one analyst describes their process in our logic, it will help other analysts understand and reproduce that process. Furthermore, one challenge in security is a justified secrecy among allies, which inhibits communication. A logic allows the analyst to abstract away from some sensitive system details.

Clarification of an analyst's own thinking is another benefit. Expressing one's reasoning in such a logical language forces an analyst to be precise. As [10] identifies, human cognitive biases often subtly insert themselves into analytic thinking. By specifying reasoning explicitly, we can examine the reasoning process for such instances of bias and work to reduce it.

Decision-support is an ultimate aim. We believe logics are a better tool for explanations than machine learning. And explanations are ultimately what scientists seek to make the unknown intelligible [7]. The components of a scientific explanation are outlined in [29]. Logical tools move us towards a scientific incident analysis in part because they can represent such explanations. The point of going through the pain of specifying a logic, rather than remaining in the realm of philosophy of science and natural language descriptions of incident analysis, is that logics are automatable. Automation is a clear prerequisite for any decision-support in a field like incident analysis, where data volumes are so large. At the same time, we have adapted logical tools that have demonstrated scalable reasoning in other contexts [16, 22]. The design of our logic is not just tailored to incident analysis, but, insofar as is possible at this stage, tailored to a scalable automation of support for incident analysis.

Based on analyst accounts and case studies, we have developed logical tools for incident analysis. Our goal is both descriptive and prescriptive. We have sought a useful and accurate description of what analysts do. At the same time, analysts should emulate these descriptions and build on them, to express their process methodically. Of course, this process will be gradual. Logical tools provide a new paradigm which helps enable this gradual advancement, alongside existing incident management and forensics practices.

Our work begins an approach to decision support for incident analysts. What we have provided so far also serves to highlight where additional formal definitions are appropriate (e.g., see Sect. 2.6). And of course, as with Separation Logic, the devil will be in the details of implementing such formal definitions [18]. Although the core sense-making and goal-setting aspects likely will remain a distinctly human endeavor, our developments provide hope that logical tools tailored to incident analysis could reduce the analyst's workload.

Acknowledgements. Spring is supported by University College London's Overseas Research Scholarship and Graduate Research Scholarship. Thanks to Simon Docherty for discussion and constructive comments.

References

1. Alberts, C., Dorofee, A., Killcrece, G., Ruefle, R., Zajicek, M.: Defining incident management processes for CSIRTS: a work in progress. Technical report. CMU/SEI-2004-TR-015, Software Engineering Institute, CMU 2004 (2004)
2. Bergman, M., Paavola, S.: 'Abduction': Term in The Commens Dictionary: Peirce's Terms in His Own Words. New Edition, 14 July 2016. http://www.commens.org/dictionary/term/abduction
3. Brotherston, J., Villard, J.: Sub-classical Boolean bunched logics and the meaning of par. In: Proceedings of CSL, vol. 24, pp. 325–342. LIPIcs (2015)
4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011)
5. Caltagirone, S., Pendergast, A., Betz, C.: The diamond model of intrusion analysis. Technical report, Center for Cyber Intelligence Analysis and Threat Research (2013). http://www.threatconnect.com/methodology/diamond_model_of_intrusion_analysis
6. Casey, E.: Digital Evidence and Computer Crime: Forensic Science, Computers, and The Internet. Academic press, Cambridge (2000)
7. Dear, P.: The Intelligibility of Nature: How Science Makes Sense of the World. University of Chicago Press, Chicago (2006)
8. Galmiche, D., Méry, D., Pym, D.: The semantics of BI and resource tableaux. Math. Struct. Comp. Sci. **15**(06), 1033–1088 (2005)
9. Henderson, L.: The problem of induction. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Summer 2018 edn. (2018)

10. Heuer, R.J.: Psychology of Intelligence Analysis. US Central Intelligence Agency (1999)
11. Horneman, A.: How to think like an analyst, 17 July 2017. https://insights.sei.cmu.edu/sei_blog/2017/07/how-to-think-like-an-analyst.html
12. Hutchins, E.M., Cloppert, M.J., Amin, R.M.: Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. Lead. Issues Inform. Warfare Secur. Res. **1**, 80 (2011)
13. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Principles of Programming Languages, pp. 14–26. ACM, London (2001). https://doi.org/10.1145/360204.375719
14. Kent, K., Chevalier, S., Grance, T., Dang, H.: Guide to integrating forensic techniques into incident response. Technical report, SP 800–86, National Institute of Standards and Technology, August 2006
15. Lamport, L.: What good is temporal logic? In: Mason, R. (ed.) IFIP Congress, pp. 657–668. Elsevier (1983)
16. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Wesley, Boston (2002)
17. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York (1992). https://doi.org/10.1007/978-1-4612-0931-7
18. O'Hearn, P.W.: From categorical logic to Facebook engineering. In: Logic in Computer Science (LICS), pp. 17–20. IEEE (2015)
19. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symbolic Logic **5**(2), 215–244 (1999)
20. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: a logic-based network security analyzer. In: USENIX Security Symposium, pp. 113–128 (2005)
21. von Plato, J.: The development of proof theory. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Winter 2016 edn. (2016)
22. Pym, D., Spring, J.M., O'Hearn, P.: Why separation logic works. Philosophy and Technology (2018).https://doi.org/10.1007/s13347-018-0312-8
23. Pym, D.J., O'Hearn, P.W., Yang, H.: Possible worlds and resources: the semantics of BI. Theor. Comput. Sci. **315**(1), 257–305 (2004)
24. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Logic in Computer Science, pp. 55–74. IEEE (2002)
25. Roesch, M.: Snort: lightweight intrusion detection for networks. In: Large Installation Systems Admin, pp. 229–238. USENIX, Seattle, November 1999
26. Shirey, R.: Internet Security Glossary, Version 2. RFC 4949, August 2007
27. Spring, J.M., Hatleback, E.: Thinking about intrusion kill chains as mechanisms. J. Cybersecur. **3**(3), 185–197 (2017)
28. Spring, J.M., Illari, P.: Review of human decision-making during incident analysis. Under review (2018)
29. Spring, J.M., Moore, T., Pym, D.: Practicing a science of security: a philosophy of science perspective. In: New Security Paradigms Workshop, Santa Cruz, 1–4 October 2017
30. Stoll, C.: The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage. Pan Books, London (1989)
31. Tambe, M.: Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned. Cambridge University Press, Cambridge (2011)