Formal Verification of Necessary and Sufficient Evidence in Forensic Event Reconstruction

Jan Gruber, Merlin Humml, Lutz Schröder and Felix C. Freiling*

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, 91058, Erlangen, Germany

ARTICLE INFO

Keywords:
digital evidence
digital investigations
forensic event reconstruction
formal methods
forensic computing

ABSTRACT

We study the problem of digital forensic event reconstruction, i.e., the question of whether a certain event has happened in the past of an execution by a given digital system. Instead of devising new search algorithms to solve the problem directly, we define two novel concepts in standard linear-time temporal logic and use these concepts to solve event reconstruction using established tools for model checking. The first concept is that of *sufficient evidence*, i.e., a characterization of states whose observation is sufficient to prove that a certain event happened in the past. The second concept is that of *necessary evidence*, i.e., a characterization of states whose negation can be used to refute the claim that a certain event happened in the past. Using the model checker NuSMV, we built a prototype that can calculate these two sets for a given digital system in order to solve the forensic event reconstruction problem. We relate these concepts to previous work in formal event reconstruction and apply it to Gladyshev's "ACME Manufacturing" benchmark example to illustrate the usefulness of our approach and the improved notion of digital evidence.

1. Introduction

"What happened and who did it?"—questions regarding the course of the deed and the perpetration are central to any forensic investigation. In the investigation of digital systems, the question in fact becomes more well-posed and specific because all the activities of a digital system are completely defined by the program, the machine model, and the user input. So at least in theory, event reconstruction should be possible, and it is natural to resort to formal methods as known from software and hardware verification to solve the arising reconstruction problems in a mathematically well-founded way.

1.1. Related Work

Gladyshev and Patel (2004) formulate the *event reconstruction problem* as follows: Using expert knowledge of a digital system, determine all possible sequences of events that have previously happened within the system from its final state and available clues of the system's behavior in the past. Digital systems are represented as finite state machines (FSMs), and evidence is formalized in the shape of *evidential statements*, which are defined as series of (assumed) observations. Actual event reconstruction is accomplished via *backtracing*, i.e., computing all possible computations leading to the state in question. The main challenge arising from this concept is the extremely large number of computations to consider even for fairly small systems. (Indeed, the number of even just the loop-free computations is exponential in the number of states, which in turn is already exponential in the number of variables, a phenomenon known as the state explosion problem).

To avoid the need to enumerate all computations of a system, it is natural to resort to temporal logics for the specification of investigative goals, and use model checking for their verification. This removes roughly one exponential layer from the theoretical complexity (leaving 'only' the state explosion problem). In an early approach of this kind, Rekhis and Boudriga (2005) introduce a dedicated extension S-TLA of the *Temporal Logic of Actions* TLA (Lamport, 1994), and discuss possible applications to reason investigative scenarios. Soltani and Hosseini-Seno (2019) formalize event reconstruction in a modal μ-calculus, with model checking performed within the mCRL2 tool set; they propose to address the state explosion problem by exploiting structural symmetries. The approach, compared to ours in more detail in Section 7, is demonstrated on a simplified model of the FAT file system.

Building upon the specific formalization of Gladyshev and Patel (2004), James et al. (2009) propose to transform the FSM model into a deterministic finite automaton (DFA) that encodes the set of system computations as a formal language, i.e., a set of strings. In order to answer an investigative question, they convert witness statements into regular expressions and eventually into further DFAs, and use them to spot conflicting statements. Technically, this approach is based on taking products of DFAs to check consistency. Like model-checking-based approaches, it avoids the doubly exponential complexity of the original backtracing approach and actually is quite related to the use of LTL model checking as pursued in the present paper, as LTL model checking is also based on translating formulae into a suitable form of automata. An advantage of the use of LTL as advocated in our work is the more compact and readable mode of expression it offers in comparison to writing automata directly, and indeed the translation from LTL into automata incurs

*Corresponding author
Email addresses: jan.gruber@fau.de (J. Gruber); merlin.humml@fau.de (M. Humml); lutz.schroeder@fau.de (L. Schröder); felix.freiling@fau.de (F.C. Freiling)
ORCID(s): 0000-0003-1862-2900 (J. Gruber); 0000-0002-2251-8519 (M. Humml); 0000-0002-3146-5906 (L. Schröder); 0000-0002-8279-8401 (F.C. Freiling)

exponential blowup (Huth and Ryan, 2004). Also, the use of modern symbolic model checkers to some degree alleviates the exponential dependence of the state space on the number of variables, which, contrastingly, remains in full force in a direct automata-theoretic approach.

To avoid the state explosion problem, Dewald (2015) formulated the specific reconstruction problem (SRP), which is geared toward the question of whether a *specific* event or action has occurred. To solve the SRP, Dewald defined the concept of *characteristic evidence (CE)*. Intuitively, *CE* of an action in regard to a set of other actions are those traits that are left by this particular action and none of the others. Therefore, the discovery of *CE* is sufficient to prove that an action has occurred, but the absence of *CE* proves nothing. To do so, he used simple set calculations that are computationally feasible. However, this comes at the price of losing precision in that there are wide-spread examples of sufficient evidence that cannot be detected using *CE*.

1.2. Contribution

In the present paper, we employ the established formalism of linear-time temporal logic (LTL) to develop a new approach to forensic event reconstruction for systems modelled as finite-state transition systems. The approach distinguishes between two different classes of evidence of specific actions: sufficient evidence and necessary evidence. Intuitively, *sufficient evidence* of an action σ is a state predicate whose observation guarantees that σ has actually happened before reaching the current state. Conversely, *necessary evidence* of an action σ is a state predicate that is always and persistently observed after occurrence of σ, so that its negation can be used to refute the claim that σ has happened. By defining these classes, we provide general notions of *forensic reconstructability* that do not seem to be explicit in previous work.

Besides offering general insight, our approach also has other (more practical) advantages: After calculating the above evidence sets, the actual checking of whether an event has occurred or not basically boils down to checking a state predicate on the final state q, which is computationally easy. In a sense, our approach therefore factors out the computational complexity of the analysis problem into the calculation of these evidence sets for specific actions.

Since we formalize the descriptions of necessary and sufficient evidence in LTL, we open the solution space enabling the use of highly optimized tools to calculate these evidence sets and thus profit from decades of research in this area (mostly within the formal verification community). We demonstrate this by utilizing the symbolic model checker NuSMV to calculate these evidence sets for Gladyshev's "ACME Manufacturing" benchmark example using a prototypical implementation that we provide as open-source software.[1]

[1] https://github.com/jgru/evidential-calculator.

1.3. Outline

The rest of the paper is structured as follows: To begin with, we provide background on the formal concepts involved (Section 2). Next, we revisit Dewald's specific reconstruction problem (SRP) and illustrate its limitations in Section 3. We then describe our approach to reasoning about evidence using linear-time temporal logic in Section 4. In Section 5, we describe our implementation. Afterward, we apply our method to a case study that has been the subject of previous publications in the field of forensic event reconstruction and illustrate the benefits of our approach in Section 6 before we provide further discussion in Section 7 and conclude in Section 8.

2. Background

We give a brief introduction to LTL, model checking, and Dijkstra's notation of guarded commands, which we use to describe programs.

2.1. Linear-time Temporal Logic

Temporal logic is a formalism to describe and reason about systems in terms of time. In LTL (Pnueli, 1977), the nature of time is considered to be linear, i.e., the basic concept is to model time as a sequence of states, so-called *computation paths*, that describe the evolution of a system over (discrete) time (for more information we recommend the textbook by Huth and Ryan (2004)). We briefly recall the syntax and semantics of LTL.

The logic is parametrized over a choice of a set `Atoms` of atomic facts of relevance for the underlying system and the task at hand. In a forensic context, such atoms might be, for example, *'mtime of* `/etc/shadow` *has been changed'*, *'socket descriptor 0xEF has been closed'*, or on some other apt abstraction level even propositions such as *'the email has been sent'*. Formulae of the logic are evaluated over a transition system that is a (simplified) model $\mathcal{M}$ of the underlying real-world system; we refer to $\mathcal{M}$ as a *finite-state transition system*, or briefly as a *model*. Formally, $\mathcal{M}$ is a triple $\mathcal{M} = (S, \rightarrow, L)$, where S is a set of *states*, $\rightarrow$ is a binary relation on S indicating *transitions* between states, and L is a labelling function $L : S \rightarrow \mathcal{P}(\texttt{Atoms})$, which assigns to each state s the set of atomic facts true at s. A *(computation) path* of $\mathcal{M}$ is then a sequence $s_1, s_2, s_3, \ldots$ of states such that $s_i \rightarrow s_{i+1}$ for all $i \geq 1$.

LTL provides a language in which to describe sets of computation paths of $\mathcal{M}$ in a rather compact and intuitive way using temporal operators. The simplest LTL formulae are atomic facts $p \in \texttt{Atoms}$. Such a formula is true for all computation paths where p is true in the first state. Temporal operators can be then used to describe the evolution of states in a computation path. For example, a formula of the shape $\Box p$ means that p must be true in the current and all future states of a computation path.

Formally, the syntax of LTL (as we use it in this paper) is given by the Backus Naur form

$$\phi, \psi ::= \bot \mid p \mid \neg\phi \mid \phi \wedge \psi \mid \bigcirc\phi \mid \Box\phi \mid \phi \,\mathcal{R}\, \psi$$

where, as indicated above, p ranges over `Atoms`. The semantics of the logic is given by saying which paths $\pi = s_1, s_2, s_3, \ldots$ *satisfy* which formulae (in which case we also say that the formula *holds* for the path). Along π, we move into the future by taking suffixes of π: For $i \geq 1$, we denote by π^i the suffix $s_i, s_{i+1}, s_{i+2}, \ldots$ of π (in particular, $\pi^1 = \pi$). The interpretation of the propositional operators $\bot, \neg, \wedge$ is standard; for instance, no path satisfies $\bot$, and a path satisfies $\neg\phi$ if it does not satisfy ϕ. The semantics of the main temporal operators is given as follows.

- $\bigcirc\phi$ holds for a path π if ϕ holds in the *next* state, i.e., for π^2.
- As mentioned above, $\Box\phi$ holds for π if ϕ holds in the present and all future states of π, i.e., for all suffixes π^i.
- $\phi \mathcal{R} \psi$ states that the property ψ must be true until and including the point in time when ϕ becomes true, so that ϕ *releases* ψ. (If ϕ never happens, then ψ is never 'released'.) Formally, $\phi \mathcal{R} \psi$ holds for π if either ψ holds for all suffixes π^i, or there is $i \geq 1$ such that ϕ holds for π^i, and for all $j \leq i$, ψ holds for π^j.

Further propositional connectives $\vee, \rightarrow$ and $\top$ are defined from $\neg, \wedge, \bot$ in the usual way. Moreover, one can define further temporal operators *eventually* and *until* by duality from the above operators; it happens that we will only require $\bigcirc$, $\Box$ and $\mathcal{R}$.

Finally, we have a notion of a formula ϕ being true in a *state* q_0 (rather than a computation path) in a finite-state transition system: ϕ holds at q_0 if ϕ holds for *all* paths that start at q_0. Note that at the level of states, saying that $\neg\phi$ holds at q_0 is *not* equivalent to saying that ϕ does not hold at q_0: The former means that no path starting at q_0 satisfies ϕ, while the latter means that not all paths starting at q_0 satisfy ϕ.

2.2. Model Checking

The process of checking whether ϕ holds at q_0 is known as *model checking* (Baier and Katoen, 2008)—a term also applied more widely to checking satisfaction of formulae in other logical formalisms. Model checking is a technique that was originally developed in the domain of formal verification of hardware to determine whether a state machine meets a given specification. Nowadays, model checking is applicable for software programs as well. Numerous model checkers for various formalisms, like SPIN, TLA+/TLC, and NuSMV, have been released as open-source software. In our present approach, we use NuSMV (Cimatti et al., 2002) to model-check LTL formulae over finite-state transition systems.

2.3. Guarded Commands

Guarded commands are a programming notation proposed by Dijkstra (1975) to facilitate correctness arguments. In essence, it is a compact notation to specify finite-state transition systems, and can thus also be used to reason about parallel programs (Chandy and Misra, 1989). We briefly introduce the notation using the program shown in Lst. 1.

Variables: $\{a, b\}$
Initial state: $\{a = 0, b = 0\}$
Actions:
`a0`: $a = 0 \longrightarrow a := 1$
`b0`: $b = 0 \longrightarrow b := 1$

Listing 1: Example program to illustrate the Guarded Commands Language.

The state space of the program is defined by an initial set of variables that store values from a specific domain. For simplicity, we use Boolean values 0 and 1 as the range of all variables in this paper. The second line defines the initial state of the program by assigning a specific value to each variable.

The program is formulated as a set of actions which each consist of a *name*, a *guard* and a *command*. The name (e.g., `a0`) is merely used to refer to specific actions and is separated from the guard with a colon. The guard is a boolean state predicate (e.g., $a = 0$) and the command is an assignment of values to variables (e.g., $a := 1$). Multiple assignments within a command are executed in parallel. Guard and command are separated by an arrow. An empty guard stands for the predicate *true*.

The set of actions defines the state transition relation of the program in the following way: If the guard of an action evaluates to true in a given state, we say that the action is *enabled* in that state. For a given state, the set of all enabled actions defines the set of possible next states. During an execution of the program, one enabled action is nondeterminstically chosen and the command of that action is executed resulting in the next state.

The program shown in Lst. 1 has two variables a and b and two actions `a0` and `b0`. In the initial state, both actions are enabled resulting in a nondeterminstic choice of which action is executed. In this example, each action disables itself by falsifying its guard. So if either `a0` or `b0` is executed first, the remaining other action is then executed. In the final state where $a = 1$ and $b = 1$ holds, no action is enabled anymore,

If the range of the variables is bounded, every such program effectively defines a finite-state transition system, with values of variables encoded by sufficiently many atoms. To facilitate the discussion, we let this system include atoms that record the action that has been taken in the previous step, on the understanding that these atoms are not part of the real system and hence cannot be used as evidence, which instead can observe only the effect of the actions on the variables. De facto, we thereby encode the actions, represented by their edge labels, into the state where their respective effect unfolds.

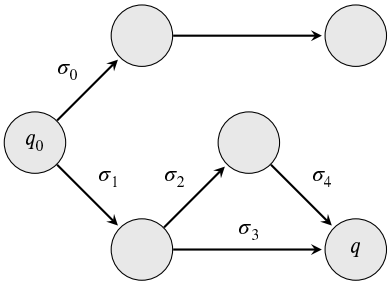

Figure 1: Illustration of the specific reconstruction problem (SRP) according to Dewald (2015).

3. The Specific Reconstruction Problem and the Problems of Characterisic Evidence

"Forensic science is reasoning backwards from effect to cause"—in this way, Berger (2010) described the main concern of the discipline in a concise form. Put differently, an elementary task in any investigation is to reconstruct past events related to the deed based on the available facets, i.e., the perceivable parts of traces produced by events (Jaquet-Chiffelle and Casey, 2021), as precisely as possible.

3.1. The Specific Reconstruction Problem

Inspired by the ideas of Gladyshev and Patel (2004), Dewald (2015) has formalized the problem of forensic event reconstruction as the specific reconstruction problem (SRP): Given an initial state q_0 and an observed state q in a finite-state transition system, determine whether or not a specific action $\sigma \in \Sigma$ necessarily happened before reaching q, in the sense that every computation path that starts in q_0 and ends in q must contain σ.

The SRP is illustrated in Figure 1, which graphically depicts a finite-state transition system with states as circles and state transitions as labelled edges between states, where the labels indicate the actions that induce the transition (recall from Section 2.3 that the actions are represented as atoms in the formal model, which however do not feature in the real system). Assuming that the system is acquired in state q, an instance of the SRP would be to ask whether action σ_0 or action σ_1 happened in the past. The answers to these questions can be easily derived from looking at the graph: While σ_0 definitely did not occur on the way to q, σ_1 definitely occurred because there is no path from q_0 to q on which σ_1 does not happen. The SRP regarding action σ_2 is not so easy to answer since q can be reached with or without executing that action. This observation shows that there are always three possible answers regarding the SRP and some action σ: (1) yes, σ definitely happened on the way to q, (2) no, σ did not happen on the way to q, and (3) σ may or may not have happened on the way to q.

3.2. Dewald's Characteristic Evidence

To solve the SRP, Dewald (2015) defines the concept of *characteristic evidence (CE)*. As mentioned above, *CE* of an action σ with respect to a set Σ of other actions are those values of variables that are left only by σ and by no other action in Σ. The discovery of *CE* in q is sufficient to prove that σ has occurred. Formally, $CE(\sigma, \Sigma)$ is the state predicate defined by all assignments of σ that are not performed by any other action in Σ.

For example, in Lst. 1, *CE* of action `a0` is the condition $a = 1$ since no other action sets a to that value. Observing $a = 1$ implies that `a0` has happened. Similarly, *CE* of action `b0` is $b = 1$ and observing that value implies that `b0` previously occured.

The concept of *CE* has the advantage of being easy to calculate. Several case studies exist that perform event reconstruction using this approach with filesystem metadata (Kälber et al., 2013, 2014) and entries from log files (Latzo and Freiling, 2019), which illustrate the principal applicability of this method in practice.

3.3. Incompleteness of the Characteristic Evidence Method

Dewald's approach draws its simplicity from ignoring the states of the system that are visited during a given execution. This is also the reason why *CE* is not a complete characterization of whether or not an action has been executed. Furthermore, the set of *CE* becomes smaller the larger the set Σ of other actions is, but a large set of compared actions increases the precision of the concept. Overall, the concept of *CE* does not identify all conditions that can be used to conclude that a certain action must have happened in the past, as we now illustrate using two examples.

Variables: $\{a, b\}$
Initial state: $\{a = 0, b = 0\}$
Actions:
`a0`: $a = 0 \longrightarrow b := 1$
`a1`: $a = 1 \longrightarrow b := 1$

Listing 2: Unreachable Action.

Consider the program in Lst. 2 where action `a1` is never executed because a is never set to 1 (`a1` is unreachable). If we compute *CE* for action `a0` with respect to $\Sigma = \{$`a1`$\}$, we observe that `a0` and `a1` have the same effect and therefore cannot be distinguished in this respect. The formal calculation of *CE* for either action results in an empty set. However, if $b = 1$ is observed, it is clear that only `a0` could have been executed since `a1` is unreachable.

The second example is shown in Lst. 3 where actions `b0` and `b1` can only be executed if $a = 1$ is true, i.e., if action `a1` has been executed before. The calculation of *CE* regarding `a1` compared to $\Sigma = \{$`a0`, `b0`, `b1`$\}$ observes only the immediate effect of `a1` and results in the condition $a = 1$.

But while in fact the observation of $a = 1$ can be used to conclude that `a1` has happened in the past, this is not the only condition to allow this. Since action `b1` is dependent on action `a1`, the observation of `b1` ($b = 1$) implies that both `b1` and `a1` have been executed before, which again illustrates that *CE*-sets are not complete.

Variables: $\{a, b\}$
Initial state: $\{a = 0, b = 0\}$
Actions:

a0:	$\longrightarrow$	$a := 0$
a1:	$\longrightarrow$	$a := 1$
b0: $a = 1$	$\longrightarrow$	$b := 0$
b1: $a = 1$	$\longrightarrow$	$b := 1$

Listing 3: Action guarded by semaphore.

4. Necessary and Sufficient Evidence

We now present a formally complete and practically more widely usable notion of "useful evidence" to solve the SRP and formulate these using LTL formulae. Figure 2 shows a rough graphical visualization of what it means to solve the SRP: Specifically, it suffices to establish that the target action must have occurred in the computation by means of which the observed state q was reached from the initial state q_0; by use of well-developed formal verification techniques, the necessity of going through all possible computation paths or even to enumerate the entire state space explicitly may be avoided.

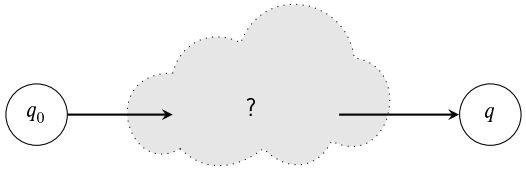

Figure 2: Graphical visualization of the SRP; q_0 denotes the initial state and q the observed state. Finding out whether action σ happened between q_0 and q does not necessarily involve explicitly enumerating all computation paths or even all states of the system.

To solve the SRP, we need a characterization of conditions on q that allow concluding that some target action σ has previously happened. Such conditions are formalized as state predicates E which represent *evidence*. If E is true in some state q, we say that q *contains* or *provides* evidence E.

4.1. Sufficient Evidence

We now turn to the first type of evidence that is useful for event reconstruction. Intuitively, *sufficient evidence* of an action σ is a state predicate (evidence) SE such that observing that predicate implies that σ has previously happened. This includes *any* state that is only reached after σ has happened, i.e., not only those states that are an immediate effect of executing σ itself, but also those states that result from follow-up actions that are guarded by σ. The only restriction is that these states cannot be reached unless σ has happened. Since we are assuming that the poststate of the system is observed statically, we may restrict sufficient evidence formulae to be purely propositional, i.e., to consist only of atoms and propositional operators ($\bot, \neg, \wedge$).

We formalize the property of *sufficient evidence (SE)* being sufficient evidence of σ as follows, using the $\mathcal{R}$ operator of LTL:

$$(\bigcirc\sigma)\,\mathcal{R}(\neg SE) \tag{1}$$

(1) holds for state predicates SE that are "released" by the action σ. The formula thus says that a state satisfying SE can only be reached after σ has been executed, which is illustrated in Fig. 3. The *next* operator $\bigcirc$ is needed because of the technical encoding of actions as atoms in poststates as mentioned in Section 2.3.

When enumerating sufficient evidence in this sense, one may optimize the list of formulae obtained by pruning conditions that are not actually satisfied in any reachable states, that is, one requires additionally that

$$\Box(\neg SE) \tag{2}$$

does *not* hold in the model (cf. the discussion in Section 2.1). We emphasize that one is interested in the weakest possible formula that still constitutes sufficient evidence; e.g., if both $a = 1$ and the disjunction $a = 1 \vee b = 1$ are SE for σ, then the latter formula is preferable as it allows establishing more easily that σ has happened.

Overall, the resulting intuition behind the concept of SE can be summed up as follows:

> "Whenever evidence SE is observable, conclude that the target action σ must have been executed."

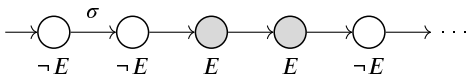

Figure 3: Visualization of a property E that constitutes sufficient evidence in the sense that $SE := E$ satisfies (1). Note that E need not actually occur immediately after $\neg E$ is released by σ (which would be one step earlier than shown in the above example) but may occur at some future point after σ is executed.

4.2. Necessary Evidence

As a counterpart to SE, we consider *necessary evidence (NE)* to be evidence which must be inevitably present in all subsequent states after the target action has been executed. The property of a formula NE being necessary evidence is formally expressed in (3), which states that except in the initial state (excluded by the "next"-statement $\bigcirc$), the execution of the target action σ implies presence of the evidence NE in all future states:

$$\bigcirc\big(\Box(\sigma \rightarrow \Box NE)\big) \tag{3}$$

This property is illustrated in Fig. 4. The intuition behind the concept of NE might be verbalized as follows:

> "Whenever the target action σ has been executed, evidence *NE* is observable in all subsequent states."

Note that, in contrast to *SE*, from the presence of *NE* one cannot draw conclusions on the execution of the target action because *NE* may already hold before σ is executed. But since *NE* must hold until the final state, one can establish that σ has *not* happened using *NE*: If *NE* is not observable in the current state, then σ has not happened (yet). In opposition to the situation with sufficient evidence, we are interested in the *strongest* possible formulae when looking for *NE*. For instance, if both $a = 1$ and the conjunction $a = 1 \wedge b = 1$ are *NE* for σ, then the latter is preferable as it allows excluding the possibility that σ has happened more easily.

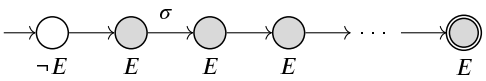

Figure 4: Visualization of a property E that constitutes necessary evidence in the sense that $NE := E$ satisfies (3). Note that E *may* hold before σ is executed, but *must* hold in every state after σ is executed; that is, all occurrences of E shown above except the first are mandated by (3).

Our notion of necessary evidence thus enhances and extends Dewald's concept of *characteristic counterevidence (CXE)*, which describes facets (values of variables) that exclude the execution of the target action: The negation of any formula constituting *CXE* is similar in spirit to necessary evidence in our sense. Like *CE*, *CXE* only considers changes directly induced by the target action, so not all necessary evidence can be obtained by negating characteristic counterevidence, as seen on separating examples similar to the ones shown above for sufficient evidence.

4.3. Action-induced Evidence

Given the definitions of *SE* and *NE* above, one can ask how these relate to Dewald's notion of *CE*. We answer this question by formalizing *action-induced evidence (AE)* in LTL. The resulting concept relates to states that are introduced by the target action itself and no other action. Formally, a formula *AE* is action-induced evidence if the following holds:

$$\begin{aligned} &\neg AE \\ &\wedge \bigcirc\Box(\sigma \rightarrow AE) \\ &\wedge \Box\big(\textstyle\bigwedge_{\sigma' \in \Sigma \setminus \{\sigma\}} (\neg AE \rightarrow \bigcirc(\sigma' \rightarrow \neg AE))\big) \end{aligned}$$

(the big conjunction symbol $\bigwedge$ expresses a finite conjunction over all actions σ' other than σ). This captures the spirit of Dewald's definition: *AE* does not hold initially, is brought about by σ, and is not brought about by any other action σ', the latter in the sense that if *AE* does not hold before execution of σ', then it does not hold afterwards either. However, we differ from Dewald's *CE* on actions that can never be executed, a situation that is ignored in the calculation of *CE*, which does not take guards into account. More concretely, in our case, such an unexecutable action would have precisely all unreachable evidence as *AE*, and hence include facets that could never be witnessed. To avoid this unintuitive (although correct) result, one could employ the same pruning as for *SE* and check that $\Box(\neg\sigma)$ does not hold in the initial state and get an empty set of *AE* instead. One sees easily that action-induced evidence is indeed sufficient evidence. On the other hand, we have seen above the converse implication does not hold, i.e., sufficient evidence need not be action-induced. The relations among some of the mentioned evidence classes are graphically summarized in Fig. 5.

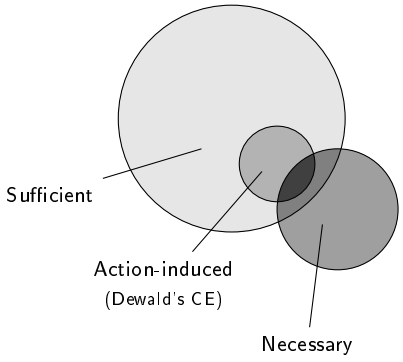

Figure 5: Venn diagram of the classes of evidence, illustrating their mutual relations. Most notably, the notion of sufficient evidence is strictly broader than that of action-induced evidence. Necessity of evidence is orthogonal to these classes.

4.4. Examples

To illustrate the above concepts, let us look at some examples.[2] The program in Lst. 1 is one of the simplest cases. Here, variables a and b are "witness" variables for the execution of `a0` and `b0`, respectively: They have value 1 if and only if that action was executed. So for both actions, *SE*, *NE* and *AE* are essentially the same (up to unnecessary strengthening or weakening, respectively), namely the conditions that $a = 1$ (for `a0`) or $b = 1$ (for `b0`), respectively.

For Lst. 2, there is no characteristic, i.e., action-induced, evidence in the sense of Dewald for `a0`, as the (unreachable) action `a1` has the same effect as action `a0`. On the other hand, the condition $b = 1$ is both *SE* and *NE* for `a0`.

Considering Lst. 3, we observe that the condition $a = 1$ is *AE* for `a1`, but the weaker condition $a = 1 \vee b = 1$ is *SE* for `a1`. Since both a and b can switch between 0 and 1 unboundedly often, there is no *NE* for any action.

A more complex example is shown in Lst. 4. The program has four variables and four actions. Like in Lst. 1, the actions have a specific variable which they exclusively set to 1. The action-induced evidence (here corresponding to Dewald's characteristic evidence) containing only the

[2] A presentation of the examples in literate programming style can be found under https://github.com/jgru/evidential-calculator/tree/master/examples

Table 1
AE set of Lst. 4.

Action	Condition
a0	$a = 1$
a1	$b = 1$
a2	$c = 1$
a3	$d = 1$

Table 2
SE set of Lst. 4.

Action	Condition
a0	$a = 1 \vee b = 1 \vee c = 1 \vee d = 1$
a1	$b = 1 \vee c = 1 \vee d = 1$
a2	$c = 1$
a3	$d = 1 \vee (b = 0 \wedge c = 1)$

Table 3
NE set of Lst. 4.

Action	Condition
a0	$a = 1 \wedge (b = 1 \vee c = 0 \vee d = 1)$
a1	$a = 1 \wedge (b = 1 \vee d = 1)$
a2	$a = 1 \wedge c = 1 \wedge (b = 1 \vee d = 1)$
a3	$a = 1 \wedge (b = 1 \vee d = 1) \wedge (c = 1 \vee d = 1)$

Variables: $\{a, b, c, d\}$
Initial state: $\{a = 0, b = 0, c = 0, d = 0\}$
Actions:

a0: $\longrightarrow$ $a := 1$
a1: $a = 1$ $\longrightarrow$ $b := 1$
a2: $b = 1$ $\longrightarrow$ $c := 1; d := 0$
a3: $b = 1$ $\longrightarrow$ $d := 1; b := 0$

Listing 4: Example program to illustrate the evidence set calculation.

immediate effects of each action is easily calculated, with results shown in Table 1.

Sufficient evidence contains both the immediate effects of the action and the effects of subsequent actions that are guarded by the respective target action. The resulting state conditions are given in Table 2.

Values of variables are necessary evidence only if they do not change after the respective action has been executed. Since the variables b and d could change back to 0, the values of these variables are only included in combination with other variables. The resulting conditions are given in Table 3.

5. Implementation

In order to transfer the theoretical concepts presented above into practice, we have developed a prototype to calculate evidence sets automatically. Fig. 6 provides an overview of the components, inputs, and outputs of the tool. The source code of our prototypical implementation is publicly available under the GNU Lesser General Public License v3.0.[3]

5.1. Dependencies of the Prototype

Our implementation uses the established model checker NuSMV[4] (Cimatti et al., 2002). In addition, we employ the Python library PyNuSMV[5] (Busard and Pecheur, 2013) to control the model checker conveniently and provide the specifications to check.

5.2. Calculation of Evidence Sets

Our tool takes a model $\mathcal{M}$ describing the system under investigation in NuSMV's specification language and the identifier of the target action σ as inputs. Using these inputs, the program calculates and outputs the various evidence formulae. The idea of the algorithm is to enumerate all possible valuations of the variables, expressed as state predicates, and use the model checker to verify in each case whether the given state predicate is *SE* or *NE*, respectively, according to the LTL formula schemes discussed above. We phrase the algorithm as working with per-state values of variables (encoded using atoms as mentioned above). We need the notion of a *partial valuation* for a set V of variables with assigned ranges of values. Such a partial valuation is a finite conjunction of formulae of the form $a = v$ where a is a variable in V and v is a value in the range of a, with every variable mentioned at most once. In the general terminology proposed by Jaquet-Chiffelle and Casey (2021) to describe observable parts of a trace, a partial valuation can be considered a facet in this context. The negation $\neg p$ of a partial valuation p is formed by negating the formula representation of p. So the conjunction becomes a disjunction and all the equalities become inequalities. The set of all such partial valuations over V will be referred to as $\mathsf{PVal}(V)$. We require that each variable is mentioned at most once in a partial valuation; if a variable is not mentioned, its value is regarded as immaterial.

1. Load the model M into the model checker.
2. Retrieve the set V of variables from the model.
3. For each partial valuation p of the variables in V, do the following:
   (a) Form the LTL specification ϕ expressing that p belongs to the evidence class of interest w.r.t. the action σ (details are discussed below), and
   (b) Check that ϕ holds in the model; if yes, accommodate p in the corresponding evidence set as specified below.

In Step 3a, we form concretized versions ϕ of (1) and (3) using p, in a manner that depends on whether we are looking for sufficient or necessary evidence. Specifically, we write $q_0 \vDash \phi$ if the initial state q_0 of the model satisfies ϕ (this is checked in Step 3b). We denote the evidence sets computed by the algorithm by $SE(\sigma, M)$ and $NE(\sigma, M)$ respectively. These sets are formally defined as follows:

$$SE(\sigma, M) = \left\{p \mid p \in \mathsf{PVal}(V), q_0 \vDash (\bigcirc\sigma)\,\mathcal{R}(\neg p)\right\}$$

$$NE(\sigma, M) = \left\{\neg p \mid p \in \mathsf{PVal}(V), q_0 \vDash \bigcirc\left(\Box(\sigma \rightarrow \Box(\neg p))\right)\right\}$$

After their construction, these sets can be used to test hypotheses and solve the SRP. To this end, we read $SE(\sigma, M)$

[3] https://github.com/jgru/evidential-calculator.
[4] https://nusmv.fbk.eu/, v2.6.0
[5] https://github.com/LouvainVerificationLab/pynusmv, v1.0rc8.

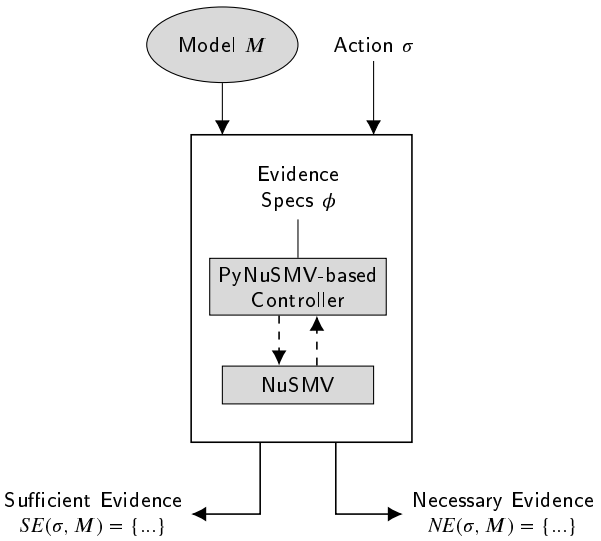

Figure 6: Overview of our prototypical implementation. The system is implemented in Python and builds on the PyNuSMV library, which provides bindings to control the model checker NuSMV. When a model M is provided in the form of NuSMV's input language, the evidence set for the specifed action σ_i will be calculated based on the LTL formulae describing the classes of evidence.

disjunctively, and $NE(\sigma, M)$ conjunctively. Note that in the latter case, the formulae contained in $NE(\sigma, M)$ are negated descriptions of valuations, so while $SE(\sigma, M)$ is effectively computed as a disjunctive normal form, $NE(\sigma, M)$ constitutes a conjunctive normal form.

We remark that the above algorithm is, of course, exponential in the number of variables; that is, it makes exponentially many calls to the model checker. This is due to the fact that the algorithm computes the optimal evidence formula, e.g., the weakest sufficient evidence. Indeed, the actual tool implements an optimization according to which small partial valations are tried first, and partial valuations extending ones that are already included in the evidence set are disregarded; this leads to more compact evidence formulae as apparent in Tables 2 and 3. Alternatively, one may just call the model checker with some target formula that is hypothesized to be sufficient evidence (for instance, a complete description of a specific observed state); in this approach, the computational cost is just that incurred by the model checker. Of course, due to the well-known state explosion problem, model checking is, in principle, already exponential in the number of variables, but modern symbolic model checkers will often perform more efficiently in practice.

6. Case Study

To illustrate the helpfulness of our methods for actual case work, we apply them to a case study that has been repeatedly discussed in previous work in the field.

```
first entry  = job from B (deleted)
second entry = job from B (deleted)
third entry  = empty
fourth entry = empty
...
nth entry    = empty
```

Listing 5: Evidence E_{obs} extracted from the print job directory of the printer in the ACME network.

6.1. The Investigation at ACME Manufacturing

Gladyshev and Patel (2004) present a fictitious example case concerning ACME Manufacturing, subsequently picked up as a case study by James et al. (2009) as well as Soltani and Hosseini-Seno (2019).

The underlying situation is described as follows: There is a local area network at the ACME Manufacturing company with two computers and a networked printer. Alice and Bob operate the network and share the costs. Alice, however, refuses to pay for maintenance of the printer and claims to have never used it. Since Bob disagrees because he once saw Alice collecting printouts, an investigation of the facts has to be initiated to resolve the dispute.

> "According to the manufacturer, the printer works as follows:
>
> 1. When a print job is received from the user it is stored in the first unallocated directory entry of the print job directory.
> 2. The printing mechanism scans the print job directory from the beginning and picks the first active job.
> 3. After the job is printed, the corresponding directory entry is marked as "deleted" , but the name of the job owner is preserved.
>
> The manufacturer also noted that
>
> 4. The printer can accept only one print job from each user at a time.
> 5. Initially, all directory entries are empty."
>
> – Gladyshev and Patel (2004, p. 4)

A forensic examination of the print job directory uncovers two processed jobs of Bob; the rest of the directory was empty, as shown in Listing 5, illustrating the observed evidence E_{obs}. However, this finding does not provide a straightforward answer to the investigative question of interest. So, what should an analyst conclude based on this finding?

6.2. Employing Our Method

In view of the description of the inner workings of the printer provided by the manufacturer, we can model the system under investigation in NuSMV's specification language. To solve the case, we then deploy our newly

Table 4
Set of sufficient evidence $SE(\texttt{add\_job\_a}, M)$. The set is read disjunctively; therefore, observing one of its elements (which are partial valuations of the variables) is sufficient to prove that action `add_job_a`, which denotes the submission of a print job to the networked printer by Alice, has happened in the past. Note that if a variable is not mentioned, its value is regarded as immaterial. Furthermore, partial valuations extending ones that are already included in the evidence set are disregarded.

	Variable	=	Value
	first entry	=	job from A
∨	first entry	=	job from A (deleted)
∨	second entry	=	job from A
∨	second entry	=	job from B
∨	second entry	=	job from A (deleted)
∨	second entry	=	job from B (deleted)

developed tool that has already been presented in Section 5. By providing the model and the action of interest—in this case, `add_job_a`—we can calculate the evidence sets. The resulting set of sufficient evidence for action `add_job_a`, which encodes the submission of a print job to the networked printer by Alice, is presented in Table 4. Referring to the set $SE_a = SE(\texttt{add\_job\_a}, M)$, we see that there is at least one element $s \in SE_a$, e.g.,

$$(\text{second entry} = \text{job from B (deleted)}),$$

that is also included in the set E_{obs} of observed evidence, which is illustrated in Lst. 5. Therefore, an investigator must draw the conclusion that Alice had printed at least once.

Of course, this fact may be validated by manual reasoning as well, as we are dealing with a simple case: As defined in the specification of the networked printer, a user can only submit one job at a time, and the entries in print job directory are used strictly in sequential order. Observing two deleted jobs of Bob implies that Bob must have submitted a print job once when there was another print job of Alice waiting to be processed.[6]

Besides these findings, we want to note that the submission of print jobs exhibits no NE in this case study since every entry in the print job directory could be potentially overwritten by follow up print jobs.

6.3. Benefits of Our Approach

Given information on the inner workings of the networked printer, we see that it is obvious that adding print jobs is conditional in two regards: First, this operation is guarded by the amount of possible print jobs per user (in this case, one) and secondly, the effect of the action (which entry in the print directory is populated) is state-dependent. In such a scenario, the method of Dewald (2015) will not be able to produce sensible results. The approach of Gladyshev and Patel (2004) can solve the case, but needs far more considerations and the introduction of specific and rather involved concepts like *evidential statements* which combine observations and a hypothesis. Moreover, their approach needs to employ a custom backtracing algorithm instead of relying on an off-the-shelf model checker.

Soltani and Hosseini-Seno (2019) solve the ACME Manufacturing case study by employing a model checker as well, in their case mCRL2, which works with an action-based branching-time temporal logic, a fixpoint extension of Hennessy-Milner logic. Their formula describing the printer investigation is, for intrinsic reasons, rather longer than ours (see Section 7 for additional comparison of the approaches).

Our approach directly translates the reconstruction problem to a computable property of the observable evidence, as shown above. The concisely stated and intuitive understanding of evidence specifies how a situation of facets has to be interpreted. Without the need to state complex interrelations, it is possible for the investigator to solve the SRP directly by looking at the acquired facets.

7. Discussion

The approach presented above aims to unify and generalize the problem of event reconstruction. We contribute to a better understanding of digital evidence, which allows precise reasoning about the quality of traces using, for the first time, linear temporal logic. In regard to event reconstruction and the SRP, the proposed method is actionable and practically usable, since it is solely concerned with observable parts of traces—the facets. Instead of abstract statements about past states of FSMs, our method puts the SRP into the center, which is regularly of principal interest in forensic analyses. By considering state, we have largely extended and improved the approach of Dewald (2015) which is limited to action-induced evidence that is calculated using set logic. To deal with the state explosion problem, we resort to an off-the-shelf symbolic model checker, which allows checking properties of a system without building the complete state graph.

We have already provided a technical comparison with work on event reconstruction via direct automata-theoretic methods (James et al., 2009) in Section 1.1. Another approach that is closely related to ours, also mentioned already in Section 1.1, uses an action-based form of the modal μ-calculus as the temporal specification language (Soltani and Hosseini-Seno, 2019). The notion of sufficient evidence remains implicit in the cited work, and necessary evidence is not considered. On a technical level, we have already noted that formulae specifying sufficient evidence in the mentioned flavour of the μ-calculus are inherently longer than our LTL formulae. This is partly due to a standard tension between labelling transitions or states; these forms of labelling are interconvertible by standard methods, which however incur blowup by a linear factor (indeed, recall that transition labels are encoded as atoms in our approach). Specifically, the formula templates given by Soltani and

[6] For further reference and illustration purposes, we present the implemented solution of the ACME Manufacturing scenario using our tool in a literate programming style at https://github.com/jgru/evidential-calculator/blob/master/examples/acme.org.

Hosseini-Seno are of linear size in the total number of actions, while our formula templates in LTL are of constant size. This is relevant insofar as model checking in either logic is (roughly) exponential in the formula size. Also, the μ-calculus is a branching-time logic, while it appears that for purposes of reconstruction of past events, linear-time formalisms that restrict attention to sequences of events, such as LTL, are inherently more suitable.

Following the strategy of separating concerns, we can swiftly determine the execution of an action once we calculated the evidence sets. In addition to that, the calculation of those sets beforehand provides clear guidance for an investigator on where to look for to prove or refute the hypothesis of the execution of a certain action, and what to conclude on which observation.

However, while this seems to be a helpful approach, there remain various challenges. A severe drawback is the high time complexity of the calculation of the evidence sets, which grows exponentially to the amount of variables (the resulting number of partial valuations) and linearly to the amount of actions (the number of sets to be computed), i.e., using only boolean variables $\mathcal{O}\left(3^{|\text{Variables}|} \times |\text{actions}|\right)$. Another hard problem is to infer apt models, i.e., state machines of the system under investigation. Currently, this involves human reasoning, a universally applicable and largely automated approach has not been proposed yet.

Furthermore, technical limitations restrict the size of processable models. In our experiments with NuSMV, we were able to only handle up to 2^{1024} states—which sounds astronomic but effectively means one can only use at most 1024 Boolean variables, illustrating that at present, there is no actual escape from the state explosion problem. In that regard, it is important to note that the results are only as good (and accurate) as the employed model itself. Thus, there is the possibility that involved parties, e.g., the defendant's lawyers, might challenge the model's correctness to undermine the conclusions.

8. Conclusion and Future Work

Event reconstruction is a salient step in every digital investigation. Over the past 20 years, various approaches working towards a formal solution have been developed, but nevertheless the problem remains unsolved in practice. From a methodical point of view, our proposed method of formalizing digital evidence with the help of LTL formulae generalizes forensic reconstructability and enables investigators to reason about an existing trace situation at a digital crime scene in a more concise way than before. It provides means to assess the meaningfulness of the evidence at hand and its inferrable implications. It defines under which circumstances which parts of the observable traces, so-called facets, can be used to reconstruct or refute past activities, and illustrates a technical realization of event reconstruction based on state predicates, thus forming a translation of the reconstruction problem and the system under investigation. In particular, we introduce clearly defined evidence classes as a precise way of communicating both the findings at hand and their significance for the case.

Future work should aim to determine suitable trace abstractions to model forensically relevant parts of reality and strive to develop automated approaches for the task of model generation. Moreover, our approach via linear-time temporal logic will in principle allow for investigating more complex behaviour, composed of more than one action. Finally, we want to explore more efficient ways of calculating evidence sets. Still, the improved notion of digital evidence and the technical realization presented can help investigators already today to answer the crucial questions of what happened and who did it.

Acknowledgements

The authors thank Stefan Milius and Lisa Rzepka for their support in the preparation of this article as well as Andreas Neubaum and the anonymous reviewers for helpful comments on the draft. Work has been supported by DFG (German Research Foundation) as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019). Merlin Humml was also supported by the DFG project RAND (grant number 377333057).

CRediT authorship contribution statement

Jan Gruber: Conceptualization, Formal Analysis, Investigation, Methodology, Software, Validation, Visualization, Writing - Original draft, Writing - Review & Editing. **Merlin Humml:** Conceptualization, Formal Analysis, Investigation, Methodology, Software, Writing - Review & Editing. **Lutz Schröder:** Conceptualization, Funding Acquisition, Formal Analysis, Supervision, Writing - Review & Editing. **Felix C. Freiling:** Conceptualization, Methodology, Project Administration, Funding Acquisition, Supervision, Writing - Review & Editing.

References

Baier, C., Katoen, J., 2008. Principles of model checking. MIT Press.

Berger, C., 2010. Criminalistics is reasoning backwards. Nederlands Juristenblad 85, 784–789.

Busard, S., Pecheur, C., 2013. Pynusmv: Nusmv as a python library, in: Brat, G., Rungta, N., Venet, A. (Eds.), NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings, Springer. pp. 453–458. doi:10.1007/978-3-642-38088-4_33.

Chandy, K.M., Misra, J., 1989. Parallel program design - a foundation. Addison-Wesley.

Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A., 2002. Nusmv 2: An open-source tool for symbolic model checking, in: Brinksma, E., Larsen, K.G. (Eds.), Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings, Springer. pp. 359–364. doi:10.1007/3-540-45657-0_29.

Dewald, A., 2015. Characteristic evidence, counter evidence and reconstruction problems in forensic computing. it Inf. Technol. 57, 339–346.

Dijkstra, E.W., 1975. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18, 453–457. doi:10.1145/360933.360975.

Gladyshev, P., Patel, A., 2004. Finite state machine approach to digital event reconstruction. Digit. Investig. 1, 130–149. doi:10.1016/j.diin.2004.03.001.

Huth, M., Ryan, M.D., 2004. Logic in computer science - modelling and reasoning about systems (2. ed.). Cambridge University Press.

James, J., Gladyshev, P., Abdullah, M.T., Zhu, Y., 2009. Analysis of evidence using formal event reconstruction, in: Goel, S. (Ed.), Digital Forensics and Cyber Crime - First International ICST Conference, ICDF2C 2009, Albany, NY, USA, September 30-October 2, 2009, Revised Selected Papers, Springer. pp. 85–98. doi:10.1007/978-3-642-11534-9_9.

Jaquet-Chiffelle, D.O., Casey, E., 2021. A formalized model of the trace. Forensic Science International 327, 110941. doi:https://doi.org/10.1016/j.forsciint.2021.110941.

Kälber, S., Dewald, A., Freiling, F.C., 2013. Forensic application-fingerprinting based on file system metadata, in: Morgenstern, H., Ehlert, R., Freiling, F.C., Frings, S., Göbel, O., Günther, D., Kiltz, S., Nedon, J., Schadt, D. (Eds.), Seventh International Conference on IT Security Incident Management and IT Forensics, IMF 2013, Nuremberg, Germany, March 12-14, 2013, IEEE Computer Society. pp. 98–112. doi:10.1109/IMF.2013.20.

Kälber, S., Dewald, A., Idler, S., 2014. Forensic zero-knowledge event reconstruction on filesystem metadata, in: Katzenbeisser, S., Lotz, V., Weippl, E.R. (Eds.), Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 19.-21. März 2014, Wien, Österreich, GI. pp. 331–343.

Lamport, L., 1994. The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16, 872–923. doi:10.1145/177492.177726.

Latzo, T., Freiling, F.C., 2019. Characterizing the limitations of forensic event reconstruction based on log files, in: 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 13th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, August 5-8, 2019, IEEE. pp. 466–475. doi:10.1109/TrustCom/BigDataSE.2019.00069.

Pnueli, A., 1977. The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society. pp. 46–57. doi:10.1109/SFCS.1977.32.

Rekhis, S., Boudriga, N., 2005. A formal logic-based language and an automated verification tool for computer forensic investigation, in: Haddad, H., Liebrock, L.M., Omicini, A., Wainwright, R.L. (Eds.), Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005, ACM. pp. 287–291. doi:10.1145/1066677.1066745.

Soltani, S., Hosseini-Seno, S., 2019. A formal model for event reconstruction in digital forensic investigation. Digit. Investig. 30, 148–160. doi:10.1016/j.diin.2019.07.006.