Gray-box Monitoring of Hyperproperties*

Sandro Stucki[1][0000−0001−5608−8273], César Sánchez[2][0000−0003−3927−4773], Gerardo Schneider[1][0000−0003−0629−6853], and Borzoo Bonakdarpour[3][0000−0003−1800−5419]

[1] University of Gothenburg, Sweden, sandro.stucki@gu.se,gerardo@cse.gu.se
[2] IMDEA Software Institute, Spain, cesar.sanchez@imdea.org
[3] Iowa State University, USA, borzoo@iastate.edu

Abstract. Many important system properties, particularly in security and privacy, cannot be verified statically. Therefore, runtime verification is an appealing alternative. Logics for hyperproperties, such as HyperLTL, support a rich set of such properties. We first show that *black-box* monitoring of HyperLTL is in general unfeasible, and suggest a *gray-box* approach. Gray-box monitoring implies performing analysis of the system at run-time, which brings new limitations to monitorabiliy (the feasibility of solving the monitoring problem). Thus, as another contribution of this paper, we refine the classic notions of *monitorability*, both for trace properties and hyperproperties, taking into account the computability of the monitor. We then apply our approach to monitor a privacy hyperproperty called *distributed data minimality*, expressed as a HyperLTL property, by using an SMT-based static verifier at runtime.

1 Introduction

Consider a *confidentiality* policy φ that requires that every pair of separate executions of a system agree on the position of occurrences of some proposition a. Otherwise, an external observer may learn some sensitive information about the system. We are interested in studying how to build runtime monitors for properties like φ, where the monitor receives independent executions of the system under scrutiny and intend to determine whether or not the system satisfies the property. While no such monitor can determine whether the system satisfies φ — as it cannot determine whether it has observed the whole (possibly infinite) set of traces — it may be able to detect violations. For example, if the monitor receives finite executions $t_1 = \{a\}\{\}\{\}\{a\}\{\}$ and $t_2 = \{a\}\{a\}\{\}\{\}\{a\}$, then it is straightforward to see that the pair (t_1, t_2) violates φ (the traces do not agree on the truth value of a in the second, fourth, and fifth positions).

* This research has been partially supported by the United States NSF SaTC Award 1813388, by the Swedish Research Council (*Vetenskapsrådet*) under Grant 2015-04154 "PolUser", by the Madrid Regional Government under Project S2018/TCS-4339 "BLOQUES-CM", by EU H2020 Project 731535 "Elastest", and by Spanish National Project PGC2018-102210-B-100 "BOSCO".

Now, if we change the policy to φ' requiring that, for every execution, there must exist a different one that agrees with the first execution on the position of occurrences of a, the monitor cannot even detect violations of φ'. Indeed, it is not possible to tell at run-time whether or not for each execution (from a possibly infinite set), there exists a related one. Such properties for which no monitor can detect satisfaction or violation are known as *non-monitorable*.

Monitorability was first defined in [27] as the problem of deciding whether any extension of an observed trace would violate or satisfy a property expressed in LTL. We call this notion *semantic black-box monitorability*. It is semantic because it defines a decision problem (the existence of a satisfying or violating trace extension) without requiring a corresponding decision procedure. In settings like LTL the problem is decidable and the decision procedures are well-studied, but in other settings, a property may be semantically monitorable even though no algorithm to monitor it exists. This notion of monitorability is "black-box" because it only considers the temporal logic formula to determine the plausibility of an extended observation that violates or satisfies the formula. This is the only sound assumption without looking inside the system. Many variants of this definition followed, mostly for trace logics [18] (see also [4]).

The definition of semantic monitorability is extended in [1] to the context of *hyperproperties* [10]. A hyperproperty is essentially a set of sets of traces, so monitoring hyperproperties involves reasoning about multiple traces simultaneously. The confidentiality example discussed above is a hyperproperty. The notion of monitorability for hyperproperties in [1] also considers whether extensions of an observed trace, or of other additional observed traces, would violate or satisfy the property. An important drawback of these notions of monitorability is that they completely ignore the role of the system being monitored and the possible set of executions that it can exhibit to compute a verdict of a property.

In this paper, we consider a landscape of monitorability aspects along three dimensions, as depicted in Fig. 1. We explore the ability of the monitor to reason about multiple traces simultaneously (the trace/hyper dimension). We first show that a large class of hyperproperties that involve quantifier alternations are non-monitorable. That is, no matter the observation, no verdict can ever be declared. We then propose a solution based on a combination of static analysis and runtime verification. If the analysis of the system is completely precise, we call it *white-box* monitoring. *Black-box* monitoring refers to the classic approach of ignoring the system and crafting general monitors that provide sound verdicts for every system. In *gray-box* monitoring, the monitor uses an approximate set of executions, given for example as a model, in addition to the observed finite execution. The combination of static analysis and runtime verification allows to monitor hyperproperties of

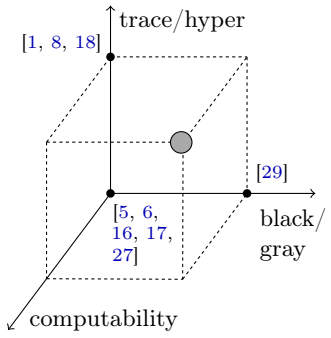

Fig. 1. The monitorability cube.

interest, but it involves reasoning about possible executions of the system (the black/gray dimension in Fig. 1). This, in turn, forces us to consider the computability limitations of the monitors themselves as programs (the computability dimension).

We apply this approach to monitoring a complex hyperproperty of interest in privacy, namely, *data minimization*. The principle of data minimization (introduced in Article 5 of the EU General Data Protection Regulation [14]) from a software perspective requires that only data that is semantically used by a program should be collected and processed. When data is collected from independent sources, the property is called *distributed data minimization* (DDM) [3, 25]. Our approach for monitoring DDM is as follows. We focus on detecting violations of DDM (which we express in HyperLTL using one quantifier alternation). We then create a gray-box monitor that collects dynamically potential witnesses for the existential part. The monitor then invokes an oracle (combining symbolic execution trees and SMT solving) to soundly decide the universally quantified inner sub-formula. Our approach is sound but approximated, so the monitor may give an inconclusive answer, depending on the precision of the static verification.

Contributions. In summary, the contributions of this paper are the following:

(1) Novel richer definitions of monitorability that consider trace and hyperproperties, and the possibility of analyzing the system (gray-box monitoring). This enables the monitoring, via the combination of static analysis and runtime verification, of properties that are non-monitorable in a black-box manner. Our novel notions of monitorability also cover the computability limitations of monitors as programs, which is inevitable once the analysis is part of the monitoring process.
(2) We express DDM as a hyperproperty and study its monitorability within the richer landscape defined above. We then apply the combined approach where the static analysis in this case is based on symbolic execution (Sect. 4).

Full proofs as well as a detailed description of our proof-of-concept implementation and its empirical evaluation can be found in the extended version of this paper [28]. The source code of our implementation is freely available online.[4]

2 Background

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the finite *alphabet*. We call each element of Σ a *letter* (or an *event*). Throughout the paper, Σ^ω denotes the set of all infinite sequences (called *traces*) over Σ, and Σ^* denotes the set of all finite traces over Σ. For a trace $t \in \Sigma^\omega$ (or $t \in \Sigma^*$), $t[i]$ denotes the i^{th} element of t, where $i \in \mathbb{N}$. We use $|t|$ to denote the length (finite or infinite) of trace t. Also, $t[i, j]$ denotes the subtrace of t from position i up to and including position j (or ϵ if $i > j$ or if $i > |t|$). In this manner $t[0, i]$ denotes the prefix of t up to and including i and $t[i, ..]$ denotes the suffix of t from i (including i).

[4] At https://github.com/sstucki/minion/

Given a set X, we use $\mathcal{P}(X)$ for the set of subsets of X and $\mathcal{P}_{fin}(X)$ for the set of finite subsets of X. Let u be a finite trace and t a finite or infinite trace. We denote the concatenation of u and t by ut. Also, $u \preceq t$ denotes the fact that u is a prefix of t. Given a finite set U of finite traces and an arbitrary set W of finite or infinite traces, we say that W extends U (written $U \preceq W$) if, for all $u \in U$, there is a $v \in W$, such that $u \preceq v$. Note that every trace in U is extended by some trace in W (we call these *trace extensions*), and that W may also contain additional traces with no prefix in U (we call these *set extensions*).

2.1 LTL and HyperLTL

We now briefly introduce LTL and HyperLTL. The syntax of LTL [26] is:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

where $a \in \mathsf{AP}$. The semantics of LTL is given by associating to a formula the set of traces $t \in \Sigma^\omega$ that it accepts:

$t \models p$	iff	$p \in t[0]$
$t \models \neg\varphi$	iff	$t \not\models \varphi$
$t \models \varphi_1 \vee \varphi_2$	iff	$t \models \varphi_1$ or $t \models \varphi_2$
$t \models \bigcirc\varphi$	iff	$t[1,..] \models \varphi$
$t \models \varphi_1 \,\mathcal{U}\, \varphi_2$	iff	for some i, $t[i,..] \models \varphi_2$ and for all $j < i$, $t[j,..] \models \varphi_1$

We will also use the usual derived operators ($\diamondsuit\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$) and ($\square\varphi \equiv \neg\diamondsuit\neg\varphi$). All properties expressible in LTL are *trace properties* (each individual trace satisfies the property or not, independently of any other trace). Some important properties, such as information-flow security policies (including confidentiality, integrity, and secrecy), cannot be expressed as trace properties but require reasoning about two (or more) independent executions (perhaps from different inputs) simultaneously. Such properties are called *hyperproperties* [10]. HyperLTL [11] is a temporal logic for hyperproperties that extends LTL by allowing explicit quantification over execution traces. The syntax of HyperLTL is:

$$\varphi ::= \forall\pi.\varphi \mid \exists\pi.\varphi \mid \psi \qquad\qquad \psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc\psi \mid \psi \,\mathcal{U}\, \psi$$

A trace assignment $\Pi : \mathcal{V} \to \Sigma^\omega$ is a partial function mapping trace variables in $\mathcal{V}$ to infinite traces. We use $\Pi_\varnothing$ to denote the empty assignment, and $\Pi[\pi \to t]$ for the same function as Π, except that π is mapped to trace t. The semantics of HyperLTL is defined by associating formulas with pairs (T, Π), where T is a set of traces and Π is a trace assignment:

$T, \Pi \models \forall\pi.\varphi$	iff	for all $t \in T$ the following holds $T, \Pi[\pi \to t] \models \varphi$
$T, \Pi \models \exists\pi.\varphi$	iff	there exists $t \in T$ such that $T, \Pi[\pi \to t] \models \varphi$
$T, \Pi \models \psi$	iff	$\Pi \models \psi$

The semantics of the temporal inner formulas is defined in terms of the traces associated with each path (here $\Pi[i,..]$ denotes the map that assigns π to $t[i,..]$

if $\Pi(\pi) = t$):

$$
\begin{array}{lll}
\Pi \models a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\
\Pi \models \psi_1 \vee \psi_2 & \text{iff} & \Pi \models \psi_1 \text{ or } \Pi \models \psi_2 \\
\Pi \models \neg\psi & \text{iff} & \Pi \not\models \psi \\
\Pi \models \bigcirc\psi & \text{iff} & \Pi[1..] \models \psi \\
\Pi \models \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \text{for some } i,\ \Pi[i,..] \models \psi_2, \text{ and for all } j < i\ T, \Pi[j,..] \models \psi_1
\end{array}
$$

We say that a set T of traces satisfies a HyperLTL formula φ (denoted $T \models \varphi$) if and only if $T, \Pi_\varnothing \models \varphi$.

Example 1. Consider the HyperLTL formula $\varphi = \forall\pi.\forall\pi'.\square(a_\pi \leftrightarrow a_{\pi'})$ and $T = \{t_1, t_2, t_3\}$, where $t_1 = \{a,b\}\{a,b\}\{\}\{b\}\cdots$, $t_2 = \{a\}\{a\}\{b\}\cdots$ and $t_3 = \{\}\{a\}\{b\}\cdots$ Although traces t_1 and t_2 together satisfy φ, t_3 does not agree with the other two, i.e., $a \in t_1(0), a \in t_2(0)$, but $a \notin t_3(0)$. Hence, $T \not\models \varphi$.

2.2 Semantic Monitorability

Runtime verification (RV) is concerned with (1) generating a monitor from a formal specification φ, and (2) using the monitor to detect whether or not φ holds by observing events generated by the system at run time. *Monitorability* refers to the possibility of monitoring a property. Some properties are non-monitorable because no finite observation can lead to a conclusive verdict. We now present some abstract definitions to encompass previous notions of monitorability in a general way. These definitions are made concrete by instantiating them for example to traces (for trace properties) or sets of traces (for hyperproperties), see Ex. 2 below.

- **Observation.** We refer to the finite information provided dynamically to the monitor up to a given instant as an *observation*. We use O and P to denote individual observations and $\mathcal{O}$ to denote the set of all possible observations, equipped with an operator $O \preceq P$ that captures the extension of an observation.
- **System behavior.** We use $\mathcal{B}$ to denote the universe of all possible *behaviors* of a system. A behavior $B \in \mathcal{B}$ may, in general, be an infinite piece of information. By abuse of notation, $O \preceq B$ denotes that observation $O \in \mathcal{O}$ can be extended to a behavior B.

Example 2. When monitoring trace properties such as LTL, we have $\mathcal{O} = \Sigma^*$, an observation is a finite trace $O \in \Sigma^*$, $O \preceq O'$ is the prefix relation on finite strings, and $\mathcal{B} = \Sigma^\omega$. When monitoring hyperproperties such as HyperLTL, an observation is a finite set of finite traces $O \subset \Sigma^*$, that is, $\mathcal{O} = \mathcal{P}_{fin}(\Sigma^*)$. The relation $\preceq$ is the prefix for finite sets of finite traces defined above. That is, $O \preceq P$ whenever for all $t \in O$ there is a $t' \in P$ such that $t \preceq t'$. Finally, $\mathcal{B} = \mathcal{P}(\Sigma^\omega)$.

We say that an observation $O \in \mathcal{O}$ *permanently satisfies* a formula φ, if every $B \in \mathcal{B}$ that extends O satisfies φ:

$$O \models^s \varphi \quad \text{iff} \quad \text{for all } B \in \mathcal{B} \text{ such that } O \preceq B,\ B \models \varphi$$

where $\models$ denotes the satisfaction relation in the semantics of the logic. Similarly, we say that an observation $O \in \mathcal{O}$ *permanently violates* a formula φ, if every extension $B \in \mathcal{B}$ violates φ:

$$O \models^v \varphi \quad \text{iff} \quad \text{for all } B \in \mathcal{B} \text{ such that } O \preceq B,\ B \not\models \varphi$$

Monitoring a system for satisfaction (or violation) of a formula φ is to decide whether a finite observation permanently satisfies (resp. violates) φ.

Definition 1 (Semantic Monitorability). *A formula φ is* (semantically) monitorable *if every observation O has an extended observation $P \succeq O$, such that $P \models^s \varphi$ or $P \models^v \varphi$.*

A similar definition of monitorability only for satisfaction or only for violation can be obtained by considering only $P \models^s \varphi$ or only $P \models^v \varphi$. Instantiating this definition of monitorability for LTL and finite traces as observations ($\mathcal{O} = \Sigma^*$ and $\mathcal{B} = \Sigma^\omega$) leads to the classic definitions of monitorability for LTL by Pnueli and Zaks [27] (see also [18]). Similarly, instantiating the definitions for HyperLTL and observations as finite sets of finite traces leads to monitorability as introduced by Agrawal and Bonakdarpour [1].

Example 3. The LTL formula $\square\diamondsuit a$ is not (semantically) monitorable since it requires an infinite-length observation, while formulas $\square a$ and $\diamondsuit a$ are monitorable. Similarly, $\forall\pi.\forall\pi.\square(a_\pi \leftrightarrow \neg a_{\pi'})$ is monitorable, but $\forall\pi.\exists\pi.\square(a_\pi \leftrightarrow \neg a_{\pi'})$ is not, as it requires an observation set of infinite size. We will prove this claim in detail in Sect. 3.

3 The Notion of Gray-box Monitoring

Most of the previous definitions of monitorability make certain assumptions: (1) the logics are trace logics, i.e. do not cover hyperproperties, (2) the system under analysis is black-box in the sense that every further observation is possible, (3) the logics are tractable, in that the decision problems of satisfiability, liveness, etc. are decidable. We present here a more general notion of monitorability by challenging these assumptions.

3.1 The Limitations of Monitoring Hyperproperties

Earlier work on monitoring hyperproperties is restricted to the quantifier alternation-free fragment, that is either $\forall^*.\psi$ or $\exists^*.\psi$ properties. We establish now an impossibility result about the monitorability of formulas of the form $\forall\pi.\exists\pi'.\square F$, where F is a state predicate. That is, F is formed by atomic propositions, a_π or $a_{\pi'}$ and Boolean combinations thereof, and can be evaluated given two valuations of the propositions from AP, one from each path π and π' at the current position. For example, the predicate $F = (a_\pi \leftrightarrow \neg a_{\pi'})$ for $\mathsf{AP} = \{a\}$ depends on the valuation of a at the first state of paths π and π'. We use v and v' in $F(v, v')$ to denote that F uses two copies of the variables v (one copy from π and another from π'). A predicate F is *reflexive* if for all valuations $v \in 2^{\mathsf{AP}}$, $F(v, v)$ is true. A predicate F is *serial* if, for all v, there is a v' such that $F(v, v')$ is true.

Theorem 1. *A HyperLTL formula of the form* $\psi = \forall\pi.\exists\pi'.\square F$ *is non-monitorable if and only if* F *is non-reflexive and serial.*

Proof (Sketch). For the "$\Leftarrow$" direction, it is easy to see that seriality implies that Σ^ω is a model of φ. Also, non-reflexivity means any observation can be extended to a non-model by adding v to every trace, so that $\neg F(v, v)$. Since every observation can be extended to a model and a non-model, φ is non-monitorable.

For the "$\Rightarrow$" direction, we prove that reflexivity or non-seriality imply monitorability. Reflexivity implies that φ is vacuously true by taking the same trace for π and π'. Then, assume non-seriality, and append to one path in the observation v such that for no v, $F(v, v')$, generating a permanent violation. □

The fragment of $\forall\exists$ properties captured by Theorem 1 is very general (and this result can be easily generalized to $\forall^+\exists^+$ hyperproperties). First, the temporal operator is just safety (the result can be generalized for richer temporal formulas). Also, every binary predicate can be turned into a non-reflexive predicate by distinguishing the traces being related. Moroever, many relational properties, such as non-interference and DDM, contain a tacit assumption that only distinct traces are being related. Seriality simply establishes that F cannot be falsified by only observing the local valuation of one of the traces. Intuitively, a predicate that is not serial can be falsified by looking only at one of the traces, so the property is not a proper hyperproperty. The practical consequence of Theorem 1 is that many hyperproperties involving one quantifier alternation cannot be monitored.

3.2 Gray-box Monitoring. Sound and Perfect Monitors

To overcome the negative non-monitorability result, we exploit knowledge about the set of traces that the system can produce (gray-box or white-box monitoring). Given a system that can produce the set of system behaviors $\mathcal{S} \subseteq \mathcal{B}$, we parametrize the notions of permanent satisfaction and permanent violation to consider only behaviors in $\mathcal{S}$:

$$O \models^s_{\mathcal{S}} \varphi \quad \text{iff} \quad \text{for all } B \in \mathcal{S} \text{ such that } O \preceq B,\ B \models \varphi$$
$$O \models^v_{\mathcal{S}} \varphi \quad \text{iff} \quad \text{for all } B \in \mathcal{S} \text{ such that } O \preceq B,\ B \not\models \varphi$$

First, we extend the definition of monitorability (Def. 1 above) to consider the system under observation.

Definition 2 (Semantic Gray-Box Monitorability). *A formula* φ *is* semantically gray-box monitorable *for a system* $\mathcal{S}$ *if every observation* O *has an extended observation* $P \succeq O$ *in* $\mathcal{S}$*, such that* $P \models^s_{\mathcal{S}} \varphi$ *or* $P \models^v_{\mathcal{S}} \varphi$*.*

In this definition, monitors must now analyze and decide properties of extended observations which is computationally not possible with full precision for sufficiently rich system descriptions.

We now introduce a novel notion of monitors that consider $\mathcal{S}$ and the computational power of monitors (the diagonal dimension in Fig. 1). A *monitor* for a

property φ and a set of traces $\mathcal{S}$ is a *computable* function $M_{\mathcal{S}} \colon \mathcal{O} \to \{\top, \bot, ?\}$ that, given a finite observation O, decides a *verdict* for φ: $\top$ indicates success, $\bot$ indicates failure, and ? indicates that the monitor cannot declare a definite verdict given only u. To avoid clutter, we write M instead of $M_{\mathcal{S}}$ when the system is clear from the context. The following definition captures when a monitor for a property φ can give a definite answer.

Definition 3 (Sound monitor). *Given a property φ and a set of behaviors $\mathcal{S}$, a monitor M is* sound *whenever, for every observation $O \in \mathcal{O}$,*

1. *if $O \models^{s}_{\mathcal{S}} \varphi$, then $M(O) = \top$ or $M(O) = ?$,*
2. *if $O \models^{v}_{\mathcal{S}} \varphi$, then $M(O) = \bot$ or $M(O) = ?$,*
3. *otherwise $M(O) = ?$.*

If a monitor is not sound then it is possible that an extension of O forces M to change a $\top$ to a $\bot$ verdict, or vice-versa. The function that always outputs ? is a sound monitor for any property, but this is the least informative monitor. A *perfect monitor* precisely outputs whether satisfaction or violation is inevitable, which is the most informative monitor.

Definition 4 (Perfect Monitor). *Given a property φ and a set of traces $\mathcal{S}$, a monitor M is* perfect *whenever, for every observation $O \in \mathcal{O}$,*

1. *if $O \models^{s}_{\mathcal{S}} \varphi$ then $M(O) = \top$,*
2. *if $O \models^{v}_{\mathcal{S}} \varphi$ then $M(O) = \bot$,*
3. *otherwise $M(O) = ?$.*

Obviously, a perfect monitor is sound. Similar definitions of perfect monitor only for satisfaction (resp. violation) can be given by forcing the precise outcome only for satisfaction (resp. violation).

A black-box monitor is one where every behavior is potentially possible, that is $\mathcal{S} = \mathcal{B}$. If the monitor uses information about the actual system, then we say it is *gray-box* (and we use *white-box* when the monitor can reason with absolute precision about the set of traces of the system). In some cases, for example to decide instantiations of a $\forall$ quantifier, a satisfaction verdict that is taken from $\mathcal{S}$ can be concluded for all over-approximations (dually under-approximations for violation and for $\exists$). For space limitations, we do not give the formal details here.

Using Defs. 3 and 4, we can add the computability aspect to capture a stronger definition of monitorability. Abusing notation, we use $O \in \mathcal{S}$ to say that observation O can be extended to a trace allowed by the system.

Definition 5 (Strong Monitorability). *A property φ is* strongly monitorable *for a system $\mathcal{S}$ if there is a sound monitor M s.t. for all observations $O \in \mathcal{O}$, there is an extended observation $P \in \mathcal{S}$ for which either $M(P) = \top$ or $M(P) = \bot$.*

A property is strongly monitorable for satisfaction if the extension with $M(P) = \top$ always exists (and analogously for violation). In what follows we will use the

term *monitorability* to refer to strong monitorability whenever no confusion may arise. It is easy to see that if a property is not semantically monitorable, then it is not strongly monitorable, but in rich domains, some semantically monitorable properties may not be strongly monitorable. One trivial example is termination for deterministic programs (that is, the halting problem). Given a prefix of the execution of a deterministic program, either the program halts or it does not, so termination is monitorable in the semantics sense. However, it is not possible to build a monitor that decides the halting problem.

Lemma 1. *If φ is strongly monitorable, then φ is semantically monitorable.*

A property may not be monitorable in a black-box manner, but monitorable in a gray-box manner. In the realm of monitoring of LTL properties, strong and semantic monitorability coincide for finite state systems (see [29]) both black-box and gray-box (for finite state systems), because model-checking and the problem of deciding whether a state of a Büchi automaton is live are decidable.

Following [8] we propose to use a combination of static analysis and runtime verification to monitor violations of $\forall^+\exists^+$ properties (or dually, satisfactions of $\exists^+\forall^+$). The main idea is to collect candidates for the outer $\exists$ part dynamically and use static analysis at runtime to over-approximate the inner $\forall$ quantifiers.

4 Monitoring Distributed Data Minimality

In this section, we describe how to monitor DDM, which can be expressed as a hyperproperty of the form $\forall^+\exists^+$. In the particular case of DDM, although we mainly deal with the input/output relation of functions and are not concerned with infinite temporal behavior, we still need to handle possibly infinite set extensions $\mathcal{S}$ for black-box monitoring. In the remainder of this section, we discuss the following, seemingly contradictory aspects of DDM:

(P1) DDM is not semantically *black-box* monitorable,
(P2) DDM is semantically *white-box* monitorable (for programs that are not DDM),
(P3) checking DDM statically is undecidable,
(P4) DDM is strongly gray-box monitorable for violation, and we give a *sound monitor*.

The apparent contradictions are resolved by careful analysis of DDM along the different dimensions of the monitorability cube (Fig. 1).

We will show how to monitor DDM and similar hyperproperties using a gray-box approach. In our approach, a monitor can decide at run time the existence of traces using a limited form of static analysis. The static analyzer receives the finite observation O collected by the monitor, but not the future system behavior. Instead it must reason under the assumption that any system behavior in $\mathcal{S}$ that is compatible with O, may eventually occur. For example, given an $\exists\forall$ formula, the outer existential quantifier is instantiated with a concrete set U of runtime traces, while possible extensions of U provided by static analysis can be used to instantiate the inner universal quantifier.

4.1 DDM Preliminaries

We briefly recapitulate the formal notion of data-minimality from [3]. Given a function $f\colon I \to O$, the problem of data minimization consists in finding a *preprocessor* function $p\colon I \to I$, such that $f = f \circ p$ and $p = p \circ p$. The goal of p is to limit the information available to f while preserving the behavior of f.

There are many possible such preprocessors (e.g. the identity function), which can be ordered according to the information they disclose, that is, according to the subset relation on their *kernels*. The kernel $\ker(p)$ of a function p is defined as the equivalence relation $(x, y) \in \ker(p)$ iff $p(x) = p(y)$. The smaller $\ker(p)$ is, the more information p discloses. The identity function is the worst preprocessor since it discloses all information (its kernel is equality — the least equivalence relation). An optimal preprocessor, or *minimizer*, is one that discloses the least amount of information.

A function f is *monolithic data-minimal* (MDM), if it fulfills either of the following equivalent conditions:

1. the identity function is a minimizer for f,
2. f is injective.

Condition 1. is an information-flow-based characterization that can be generalized to more complicated settings in a straightforward fashion. Condition 2. is a purely logical or *data-based* characterization more suitable for implementation in e.g. a monitor.

MDM is the strongest form of data minimality, where one assumes that all input data is provided by a single source and thus a single preprocessor can be used to minimize the function. If inputs are provided by multiple sources (called a distributed setting) and access to the system implementing f is restricted, it might be impossible to use a single preprocessor. For example, consider a web-based auction system that accepts bids from n bidders, represented by distinct input domains $I_1, \ldots, I_n$, and where concrete bids $x_k \in I_k$ are submitted remotely. The auction system must compute the function $m(x_1, \ldots, x_n) = \max_k\{x_k\}$, which is clearly non-injective and, hence, non-MDM. In this case, a single, monolithic minimizer cannot be used since different bidders need not have any knowledge of each other's bids. Instead, bidders must try to minimize the information contained in their bid locally, in a distributed way, before submitting it to the auction.

The problem of *distributed data minimization* consists in building a collection $p_1, \ldots, p_n$ of n independent preprocessors $p_k\colon I_k \to I_k$ for a given function $f\colon I_1 \times \cdots \times I_n \to O$, such that their parallel composition $p(x_1, \ldots, x_n) = (p_1(x_1), \ldots, p(x_n))$ is a preprocessor for f. Such composite preprocessors are called *distributed*, and a distributed preprocessor for f that discloses the least amount of information is called a *distributed minimizer* for f. Then, one can generalize the (information-flow) notion of data-minimality to the distributed setting as follows. The function f is *distributed data-minimal* (DDM) if the identity function is a distributed minimizer for f. Returning to our example, the maximum function m defined above is DDM. As for MDM, there is an equivalent, data-based characterization of DDM defined next.

Definition 6 (distributed data minimality [3, 24]). *A function f is* distributed data-minimal (DDM) *if, for all input positions k and all $x, y \in I_k$ such that $x \neq y$, there is some $z \in I$, such that $f(z[k \mapsto x]) \neq f(z[k \mapsto y])$.*

We use Def. 6 to explore how to monitor DDM. In the following, we assume that the function $f\colon I_1 \times \cdots \times I_n \to O$ has at least two arguments ($n \geq 2$). Note that for unary functions, DDM coincides with MDM. Since MDM is a $\forall^+$-property (involving no quantifier alternations), most of the challenges to monitorability discussed here do not apply [25]. We also assume, without loss of generality, that the function f being monitored has only nontrivial input domains, i.e. $|I_k| \geq 2$ for all $k = 1, \ldots n$. If I_k is trivial then this constant input can be ignored. Finally, note that checking DDM statically is undecidable (P3) for sufficiently rich programming languages [3].

4.2 DDM as a Hyperproperty

We consider data-minimality for total functions $f\colon I \to O$. Our alphabet, or set of events, is the set of possible *input-output (I/O) pairs* of f, i.e. $\Sigma_f = I \times O$. Since a single I/O pair $u = (u_{in}, u_{out}) \in \Sigma_f$ captures an entire run of f, we restrict ourselves to observing singleton traces, i.e. traces of length $|u| = 1$. In other words, we ignore any temporal aspects associated with the computation of f. This allows us to use first-order predicate logic — without any temporal modalities — as our specification logic.

DDM is a hyperproperty, expressed as a predicate over sets of traces, even though the traces are I/O pairs. The set of observable behaviors $\mathcal{O}_f$ of a given f consists of all *finite sets* of I/O pairs $\mathcal{O}_f = \mathcal{P}_{fin}(\Sigma_f)$. The set of all possible system behaviors $\mathcal{B}_f = \mathcal{P}(\Sigma_f)$ additionally includes *infinite sets* of I/O pairs.

Example 4. Let $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be the addition function on natural numbers, $f(x, y) = x + y$. Then $I = \mathbb{N} \times \mathbb{N}$, $O = \mathbb{N}$, and a valid trace $u \in \Sigma_f$ takes the form $u = ((x, y), z)$, where x, y and z are all naturals. Both $U = \{((1, 2), 3), ((2, 1), 3)\}$ and $V = \{((1, 1), 3)\}$ are considered observable behaviors $U, V \in \mathcal{O}_f$, even though V does not correspond to a valid system behavior since $f(1, 1) \neq 3$. Remember that we do not discriminate between valid and invalid system behaviors in a black-box setting.

We now express DDM as a hyperproperty, using HyperLTL, but with only state predicates (no temporal operators). Given a tuple $x = (x_1, x_2, \ldots, x_n)$, we write $\mathrm{proj}_i(x)$ or simply x_i for its i-th projection. Given an I/O pair $u = (x, y)$ we use u_{in} for the input component and u_{out} for the output component (that is $u_{in} = x$ and $u_{out} = y$). Given trace variables π, π', we define

$$\mathrm{output}(\pi, \pi') \stackrel{\text{def}}{=} \pi_{out} = \pi'_{out} \qquad \pi \text{ and } \pi' \text{ agree on their output,}$$

$$\mathrm{same}_i(\pi, \pi') \stackrel{\text{def}}{=} \mathrm{proj}_i(\pi_{in}) = \mathrm{proj}_i(\pi'_{in}) \qquad \pi \text{ and } \pi' \text{ agree on the } i\text{-th input,}$$

$$\mathrm{almost}_i(\pi, \pi') \stackrel{\text{def}}{=} \bigwedge_{k \neq i} \mathrm{proj}_k(\pi_{in}) = \mathrm{proj}_k(\pi'_{in}) \qquad \pi \text{ and } \pi' \text{ agree on all but the } i\text{-th input}$$

Example 5. Let $u = ((1,2),3)$, $u' = ((2,1),3)$, and $\Pi = \{\pi \mapsto u, \pi' \mapsto u'\}$. Then $\Pi \models \text{output}(\pi, \pi')$, but $\Pi \not\models \text{same}_1(\pi, \pi')$ and $\Pi \not\models \text{almost}_1(\pi, \pi')$.

We define DDM for input argument i as follows:

$$\varphi_i \;=\; \forall\pi.\forall\pi'.\exists\tau.\exists\tau'.\; \neg\,\text{same}_i(\pi,\pi') \rightarrow \begin{pmatrix} \text{same}_i(\pi,\tau) \wedge \text{same}_i(\pi',\tau') \wedge \\ \text{almost}_i(\tau,\tau') \wedge \neg\,\text{output}(\tau,\tau') \end{pmatrix}$$

In words: given any pair of traces π and π', if π_{in} and π'_{in} differ in their i-th position, then there must be some common values z for the remaining inputs, such that the outputs of f for $\tau_{in} = z[i \mapsto \text{proj}_i(\pi_{in})]$ and $\tau'_{in} = z[i \mapsto \text{proj}_i(\pi'_{in})]$ differ. Note that z does not appear in φ_i directly, instead it is determined implicitly by the (existentially quantified) traces τ and τ'. Finally, *distributed data minimality* for f is defined as

$$\varphi_{\mathsf{dm}} \;=\; \bigwedge_{i=1}^{n} \varphi_i.$$

The property φ_{dm} follows the same structure as the logical characterization of DDM from Sect. 4.1. The universally quantified variables range over the possible inputs at position i, while the existentially quantified variables τ and τ' range over the other inputs and the outputs. Note also that, given the input coordinates of π, π', and τ, all the output coordinates, as well as the input coordinates of τ', are uniquely determined.[5]

Example 6. Consider again $U = \{((1,2),3),((2,1),3)\}$ and $V = \{((1,1),3)\}$ from Ex. 4. Then, $V \models \varphi_{\mathsf{dm}}$ trivially holds, but $U \not\models \varphi_{\mathsf{dm}}$ because when $\Pi(\pi) \neq \Pi(\pi')$ there is no choice of $\Pi(\tau), \Pi(\tau') \in U$ for which $\Pi \models \neg\,\text{output}(\tau,\tau')$ holds.

Note that, in the above example, $V \models \varphi_{\mathsf{dm}}$ holds despite the fact that V is not a valid behavior of the example function $f(x,y) = x + y$. Indeed, whether or not $U \models \varphi_{\mathsf{dm}}$ holds for a given U is independent of the choice of f. In particular, $\Sigma_f \models \varphi_{\mathsf{dm}}$, for any choice of f regardless of whether f is data-minimal or not. This is already a hint that the notion of semantic black-box monitorability is too weak to be useful when monitoring φ_{dm}. Since Σ_f is a model of φ_{dm}, no observation U can have an extension that permanently violates φ_{dm}. As we will see shortly, gray-box monitoring does not suffer from this limitation. Monitorability of DDM for violations becomes possible once we exclude potential models such as Σ_f which do not correspond to valid system behaviors.

Remark. Note that though our definition and approach work for general (reactive) systems, the DDM example is admittedly a non-reactive system with traces of length 1. This, however, is not a limitation of the approach. Extending DDM for reactive systems is left as future work.

[5] For simplicity, even though φ_{dm} is not in prenex normal form, it is a finite conjunction of $\forall\forall\exists\exists$ formulas in prenex normal form so a finite number of monitors can be built and executed in parallel, one per input argument.

4.3 Properties of DDM

Since φ_{dm} is a $\forall^+\exists^+$ property, it should not come as a surprise that it is *not* semantically black-box monitorable in general (P1). Although DDM is not a temporal property, the proof of non-monitorability follows the same basic structure as that of Theorem 1 [28]. In particular, since $\Sigma_f \models \varphi_{\mathsf{dm}}$ for any f, no set of I/O pairs U can permanently violate φ_{dm}. In other words, φ_{dm} is clearly not black-box monitorable for violations.

However, and perhaps surprisingly, φ_{dm} is semantically white-box monitorable for violations (P2). That is, if f is not DDM, there is hope to detect it. To make this statement more precise, we first need to identify the set of valid system behaviors $\mathcal{S}_f$ of f. We define $\Sigma_f^{\#} = \{(x, y) \mid f(x) = y\}$ to be the set of I/O pairs that correspond to executions of f. Then $\mathcal{S}_f = \mathcal{P}(\Sigma_f^{\#})$ precisely characterizes the set of valid system behaviors.

Example 7. Define $g\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as $g(x, y) = x$, i.e. g simply ignores its second argument. Then $\Sigma_g^{\#} = \{((x, y), x) \mid x, y \in \mathbb{N}\}$. It is easy to show that DDM is white-box monitorable for g. Any finite set of valid traces U can be extended to include a pair of traces u, u' that only differ in their second input value, e.g. $u = ((1, 1), 1)$ and $u' = ((1, 2), 1)$. Now, consider any $T \in \mathcal{S}_f$ that extends $U \cup \{u, u'\}$. Clearly, T cannot contain any trace v for which $\mathrm{proj}_1(v_{in}) = 1$ but $v_{out} \neq 1$ as that would constitute an invalid system behavior. But T would have to contain such a trace to be a model of φ_2. Hence, $T \not\models \varphi_{\mathsf{dm}}$ for any such T, which means $U \cup \{u, u'\}$ permanently violates φ_{dm}.

Note the crucial use of information about g in the above example: it is the restriction to *valid* extensions $T \in \mathcal{S}_f$ that excludes trivial models such as Σ_f and thereby restores (semantic) monitorability for violations. The apparent conflict between (P1) and (P2) is thus resolved.

With the extra information that gray-box monitoring affords, we can make more precise claims about properties like DDM: whether or not a property is monitorable may, for instance, depend on whether the property actually holds for the system under scrutiny. Concretely, for the case of DDM, we show the following.

Theorem 2. *Given a function $f\colon I \to O$, the formula φ_{dm} is semantically gray-box monitorable in $\mathcal{S}_f$ if and only if either f is distributed non-minimal or the input domain I is finite.*

Proof (Sketch). If I is finite, $\Sigma_f^{\#} \in \mathcal{S}_f$ is a finite extension of any U and also permanently satisfies or violates φ_{dm}. If, instead, I is infinite and f is not distributed minimal, then there must be some input position i and some pair of distinct inputs $x \neq x' \in I_i$, such that $f(z[i \mapsto x]) = f(z[i \mapsto x'])$ for any choice of $z \in I$. Any set U extended by a pair of traces featuring these inputs at position i (permanently) violates φ_{dm}. The proof for the case where I is infinite and f is distributed minimal uses a similar idea to construct counterexamples to permanent satisfaction of φ_{dm}. (See our tech-report for the full proof [28].) □

Intuitively, this means that f cannot be monitored for satisfaction. Note that the semantic monitorability property established by Theorem 2 is independent of whether we can actually decide DDM for the given f. We address the question of strong monitorability later on in this section.

If I is finite, it is easy to strengthen Theorem 2 by providing a perfect monitor M_{dm} for φ_{dm}. Since f is assumed to be a total function with a finite domain, we can simply check the validity of φ_{dm} for every trace $U \subseteq \Sigma_f^{\#}$ and tabulate the result. To do so, the $\exists$ and $\forall$ quantifiers in φ_{dm} can be converted into conjunctions and disjunctions over U.

Corollary 1. *For $f : I \to O$ with finite I, φ_{dm} is strongly monitorable in $\mathcal{S}_f$.*

If I is infinite, then φ_{dm} is not semantically monitorable for satisfaction, but we can still hope to build a sound monitor for violation of φ_{dm}.

4.4 Building a Gray-box Monitor for DDM

In what follows, we assume a computable function capable of deciding DDM only for some instances. This function, that we call oracle, will serve as the basis for a *sound* monitor for DDM (P4). This monitor will detect some, but not all, violations of DDM when given sets of observed traces, thus resolving the apparent tension between (P3) and (P4).

Given $f\colon I_1 \times \cdots \times I_n \to O$, we define the predicate φ_f as

$$\varphi_f(i, x, y) = \exists z \in I.\, f(z[i \mapsto x]) \neq f(z[i \mapsto y]),$$

and assume a total computable function $N_{f,i}\colon I_i \times I_i \to \{\top, \bot, ?\}$ such that

$$N_{f,i}(x, y) = \begin{cases} \top \text{ or } ? & \text{if } \varphi_f(i, x, y) \text{ holds,} \\ \bot \text{ or } ? & \text{otherwise.} \end{cases}$$

The function $N_{f,i}$ acts as our oracle to instantiate the existential quantifiers in φ_{dm}. As discussed earlier, such oracles may be implemented by statically analyzing the system under observation (here, the function f). In our proof-of-concept implementation, we extract $\varphi_f(i, x, y)$ from f using *symbolic execution*, and use an SMT solver to compute $N_{f,i}(x, y)$ [28].

We now define a monitor M_{dm} for φ_{dm} as follows:

$$M_{\mathsf{dm}}(U) = \begin{cases} ? & \text{if } f(u_{in}) \neq u_{out} \text{ for some } u \in U, \\ ? & \text{if } \bigwedge_{i=1}^{n} \bigwedge_{u,u' \in U} N_{f,i}(\mathrm{proj}_i(u_{in}), \mathrm{proj}_i(u'_{in})) \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Intuitively, the monitor $M_{\mathsf{dm}}(U)$ checks the set of traces U for violations of DDM by verifying two conditions: the first condition ensures the *consistency* of U, i.e. that every trace in U does in fact correspond to a valid execution of f; the second condition is necessary for U *not* to permanently violate φ_{dm}. Hence, if it fails, U must permanently violate φ_{dm}. Since $N_{f,i}$ is computable, so is M_{dm}. Note

that M_{dm} never gives a positive verdict $\top$. This is a consequence of Theorem 2: if f is DDM, then φ_{dm} is not monitorable in $\mathcal{S}_f$. In other words, DDM is not monitorable for satisfaction.

The second condition in the definition of M_{dm} is an approximation of φ_{dm}: the universal quantifiers are replaced by conjunctions over the finite set of input traces U, while the existential quantifiers are replaced by a single quantifier ranging over all of $\Sigma_f^{\#}$ (not just U). This approximation is justified formally by the following theorem [28].

Theorem 3 (soundness). *The monitor M_{dm} is sound. Formally,*

1. $U \models^{s}_{\mathcal{S}_f} \varphi_{\mathsf{dm}}$ *if* $M_{dm}(U) = \top$, *and*
2. $U \models^{v}_{\mathcal{S}_f} \varphi_{\mathsf{dm}}$ *if* $M_{dm}(U) = \bot$.

4.5 Proof-of-Concept Implementation

We have implemented the ideas described above in a proof-of-concept monitor for data minimization called `minion`. The monitor uses the symbolic execution API and the SMT backend of the KeY deductive verification system [2, 19] to extract logical characterizations of Java programs (their *symbolic execution trees*). It then extends them to first-order formulas over sets of observed traces, and checks the result using the state-of-the-art SMT solver Z3 [21, 22]. The `minion` monitor is written in Scala and provides a simple command-line interface (CLI). Its source code is freely available online at https://github.com/sstucki/minion/. A detailed description of `minion`, including examples, appears in the extended version of this paper [28].

5 Related Work

LTL Monitorability. Pnueli and Zaks [27] introduced monitorability as the existence of extension of the observed traces that permanently satisfy or violate an LTL property. It is known that the set of monitorable LTL properties is a superset of the union of safety and co-safety properties [5, 6] and that it is also a superset of the set of obligation properties [15, 16]. Havelund and Peled [18] introduce a finer-grained taxonomy distinguishing between *always* finitely satisfiable (resp. refutable), and *sometimes* finitely satisfiable where only some prefixes are required to be monitorable (for satisfaction). Their taxonomy also describes the relation between monitorability and classical safety properties. This is a new dimension in the monitorability cube in Fig. 1 which we will study in the future. While all the notions mentioned above ignore the system, predictive monitoring [29] considers the traces allowed in a given finite state system.

Monitoring HyperLTL. Monitoring hyperproperties was first studied in [1], which introduces the notion of monitorability for HyperLTL [12] and gives an algorithm for a fragment of alternation-free HyperLTL. This is later generalized to the full fragment of alternation-free formulas using formula rewriting in [9], which

can also monitor alternating formulas but only with respect to a fixed finite set of finite traces. Finally, [17] proposes an automata-based algorithm for monitoring HyperLTL, which also produces a monitoring verdict for alternating formulas, but again for a fixed trace set. The complexity of monitoring different fragments of HyperLTL was studied in detail in [7]. The idea of gray-box monitoring for hyperproperties, as a means for handling non-monitoriable formulas, was first proposed in [8].

Data minimization. A formal definition of *data minimization* and the concept of *data minimizer* as a preprocessor appear in [3], which introduces the monolithic and distributed cases. Minimality is closely related to information flow [13]. Malacaria et al. [20] present a symbolic execution-based verification of non-interference security properties for the *OpenSSL* library. In our paper, we have focused on a version of distributed minimization which is not monitorable in general. For stronger versions (cf. [3]), Pinisetty et al. [24, 25] show that monitorability for satisfaction is not possible, but it is for violation. (the paper also introduces an RV approach for similar safety hyperproperties for deterministic programs).

6 Conclusions

We have rephrased the notion of monitorability considering different dimensions, namely (1) whether the monitoring is black-box or gray-box, (2) whether we consider trace properties or hyperproperties, and (3) taking into account the computatibility aspects of the monitor as a program. We showed that many hyperproperties that involve quantifier alternation are non-monitorable in a black-box manner and proposed a technique that involves inspecting the behavior of the system. In turn, this forces to consider the computability limitations of the monitor, which leads to a more general notion of monitorability.

We have considered distributed data minimality (DDM) and expressed this property in HyperLTL, involving one quantifier alternation. We then presented a methodology to monitor violations of DDM, based on a model extracted from the program being monitored in the form of its symbolic execution tree, and an SMT solver. We have implemented a tool (`minion`) and applied it to a number of representative examples to assess the feasibility of our approach [28].

As future work, we plan to extend the proposed methodology for other hyperproperties, particularly in the concurrent and distributed setting. We are also planning to use bounded model checking as our verifier at run-time by combining over- and under-approximated methods to deal with universal and existential quantifiers in HyperLTL formulas. Another interesting problem is to apply gray-box monitoring for hyperproperties with real-valued signals (e.g., HyperSTL [23]). Finally, we intend to extend the definition and results of data minimality in order to capture reactivity, and study monitorability in this setting.

References

1. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: Proc. of the IEEE 29th Computer Security Foundations (CSF'16). pp. 239–252. IEEE CS Press (2016)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer (2016)
3. Antignac, T., Sands, D., Schneider, G.: Data Minimisation: A Language-Based Approach. In: Proc. of the 32nd IFIP TC 11 Int'l Conf. on ICT Systems Security and Privacy Protection (SEC'17). IFIP Advances in Information and Communication Technology (AICT), vol. 502, pp. 442–456 (2017)
4. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Lectures on Runtime Verification, chap. Introduction to Runtime Verification, pp. 1–33. Springer (2018)
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM T. Softw. Eng. Meth. 20(4), 14 (2011)
6. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly—but how ugly is ugly? In: Proc. of the 7th Int'l Workshop on Runtime Verification (RV'07). LNCS, vol. 4839, pp. 126–138. Springer (2007)
7. Bonakdarpour, B., Finkbeiner, B.: The complexity of monitoring hyperproperties. In: CSF'18. pp. 162–174. IEEE CS Press (2018)
8. Bonakdarpour, B., Sánchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Proc. of the 8th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'2018). Verification. Part II. LNCS, vol. 11245, pp. 8–27. Springer (2018)
9. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free HyperLTL. In: Proc. of the 23rd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17). LNCS, vol. 10206, pp. 77–93. Springer (2017)
10. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security 18(6), 1157–1210 (2010)
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: In Proc. of the Third Int'l Conf on Principles of Security and Trust (POST'14). LNCS, vol. 8414, pp. 265–284. Springer (2014)
12. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: POST'14. LNCS, vol. 8414, pp. 265–284. Springer (2014)
13. Cohen, E.: Information transmission in computational systems. SIGOPS Oper. Syst. Rev. 11(5), 133–139 (1977)
14. European Commission: Proposal for a Regulation of the European Parliament and of the Council on the protection of individuals with regard to the processing of personal data and on the free movement of such data (GDPR). Tech. Rep. 2012/0011 (COD), European Commission (January 2012)
15. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of Safety-Progress properties. In: Proc. of 9th Int'l Workshop on Runtime Verification (RV'09). LNCS, vol. 5779, pp. 40–59. Springer (2009)
16. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer (STTT) 14(3), 349–382 (2012)

17. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. In: Proc. of 17th Int'l Conf. on Runtime Verification (RV'17). LNCS, vol. 10548, pp. 190–207. Springer (2017)
18. Havelund, K., Peled, D.: Runtime verification: From propositional to first-order temporal logic. In: Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18). LNCS, vol. 11237, pp. 90–112. Springer (2018)
19. KeY contributors: The KeY project. https://www.key-project.org (accessed 5 Nov 2018)
20. Malacaria, P., Tautchning, M., DiStefano, D.: Information leakage analysis of complex C code and its application to OpenSSL. In: Proc. of the 7th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'2016). Part I. LNCS, vol. 9952, pp. 909–925. Springer (2016)
21. Microsoft Research: The Z3 theorem prover. https://github.com/Z3Prover/z3 (accessed 5 Nov 2018)
22. Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). LNCS, vol. 4963, pp. 337–340. Springer (2008)
23. Nguyen, L.V., Kapinski, J., Jin, X., Deshmukh, J.V., Johnson, T.T.: Hyperproperties of real-valued signals. In: Proc. of the 15th ACM-IEEE Int'l Conf. on Formal Methods and Models for System Design (MEMOCODE'17). pp. 104–113. ACM (2017)
24. Pinisetty, S., Antignac, T., Sands, D., Schneider, G.: Monitoring data minimisation. Tech. Rep. abs/1801.02484, CoRR–arXiv.org (2018), http://arxiv.org/abs/1801.02484
25. Pinisetty, S., Sands, D., Schneider, G.: Runtime verification of hyperproperties for deterministic programs. In: Proc. of the 6th Conf. on Formal Methods in Software Engineering (FormaliSE@ICSE'18). pp. 20–29. ACM (2018)
26. Pnueli, A.: The temporal logic of programs. In: Proc. of the 18th IEEE Symp. on Foundations of Computer Science (FOCS'77). pp. 46–67. IEEE Computer Society Press (1977)
27. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Proc. of the 14th Int'l Symp on Formal Methods (FM'06). LNCS, vol. 4085, pp. 573–586. Springer (2006)
28. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties (extended version). Tech. Rep. abs/1906.08731, CoRR–arXiv.org (2019), http://arxiv.org/abs/1906.08731
29. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Proc. of 4th NASA Int'l Symp on Formal Methods (NFM'12). LNCS, vol. 7226, pp. 418–432. Springer (2012)