Runtime Verification of Hyperproperties for Deterministic Programs

Srinivas Pinisetty
University of Gothenburg
Gothenburg, Sweden
srinivas.pinisetty@gu.se

Gerardo Schneider
University of Gothenburg
Gothenburg, Sweden
gersch@chalmers.se

David Sands
Chalmers University of Technology
Gothenburg, Sweden
dave@chalmers.se

ABSTRACT

In this paper, we consider the runtime verification problem of safety hyperproperties for deterministic programs. Several security and information-flow policies such as data minimality, non-interference, integrity, and software doping are naturally expressed formally as safety hyperproperties. Although there are monitoring results for hyperproperties, the algorithms are very complex since these are properties over *set of* traces, and not over single traces. For the deterministic input-output programs that we consider, and the specific safety hyperproperties we are interested in, the problem can be reduced to monitoring of trace properties. In this paper, we present a simpler monitoring approach for safety hyperproperties of deterministic programs. The approach involves transforming the given safety hyperproperty into a trace property, extracting a characteristic predicate for the given hyperproperty, and providing a parametric monitor taking such predicate as parameter. For any hyperproperty in the considered subclass, we show how runtime verification monitors can be synthesised. We have implemented our approach in the form of a parameterised monitor for the given class, and have applied it to a number of hyperproperties including data minimisation, non-interference, integrity and software doping. We show results concerning both offline and online monitoring.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; • **Software and its engineering → Formal software verification**; • **Security and privacy → Formal methods and theory of security**;

KEYWORDS

Monitoring, Runtime Verification, Security, Information-flow

ACM Reference Format:
Srinivas Pinisetty, Gerardo Schneider, and David Sands. 2018. Runtime Verification of Hyperproperties for Deterministic Programs. In *FormaliSE '18: FormaliSE '18: 6th Conference on Formal Methods in Software Engineering , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3193992.3193995

FormaliSE '18, June 2, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5718-0/18/06…$15.00
https://doi.org/10.1145/3193992.3193995

1 INTRODUCTION

A program monitor for a given property observes the input-output actions of a program and provides, on observation of each event, a verdict as to whether the program has (not) satisfied the property, or whether it remains an open question that can (at best) be determined by future events.

Not all properties are monitorable (in the sense that there exits a computable monitor for that property), but it is straightforward to construct monitors for combinations of safety and co-safety properties. Concrete constructions have been proposed for monitors by deriving them from properties expressed as automata or linear temporal logic (LTL) [6, 12, 15].

In this paper we are concerned with information-flow properties such as non-interference – a class of properties that have been dubbed *hyperproperties* [11], defined over sets of sets of traces instead of over set of traces. As a simple example, consider the property "there is at most one possible value output by the program". Without any information about the program to be monitored, this property cannot be characterised by a single set of input-output traces, since the property does not specify a *specific* value to be output. In the terminology of Clarkson and Schneider this is a *2-safety hyperproperty* [11], meaning that it can be expressed as a universal quantification over pairs of finite traces. In this example the property can be expressed as: for all pairs of traces (of the program), the output value is the same in both traces.

Although it is not possible to construct monitors for hyperproperties in general [9], it is possible, in a certain sense, to construct monitors for safety hyperproperties [1, 9]. This may seem surprising from the perspective of enforcement of program properties by *reference monitors*, since it is known that we cannot *enforce* such properties [25]. There are essentially two reasons why monitoring is possible. Firstly, the notion of monitoring from runtime verification is different from the demands of an enforcement mechanism, since it accepts the inevitable incompleteness (the "don't know yet" answer), but still demands optimality from the black-box perspective: you should not answer "don't know" unless there is genuine uncertainty. Secondly, works on monitoring hyperproperties make the implicit assumption that multiple clones of the system can be made and monitored. This latter assumption enables the monitor to detect satisfaction or violation of properties stated in terms of multiple runs of the system. At the same time this suggests that hyperproperty monitoring is an offline testing activity rather than an online mechanism for observing the behaviour of an actual operational system. We argue that the main beneficial characteristics of online runtime verification is simplicity and low technical complexity, but that this is lost when considering monitoring of hyperproperties [1, 9]; runtime verification mechanisms become very complex when compared with runtime monitoring approaches for trace properties [6, 7].

In this paper we simplify the view of monitoring of hyperproperties by reducing it to the simple case of monitoring of trace properties, thus regaining the lost simplicity. To do this we study 2-safety hyperproperties for simple programs whose execution consists of a single event, an input output pair. Here, 2-safety hyperproperties, denoted as Hyper_{2S}, are just properties expressed with universal quantification over two traces. Though this might seem a restriction, this class includes termination insensitive[1] variants of many interesting security properties including noninteference [10], integrity [10], software doping [5, 13], and data minimality [3].

We make some modest assumptions about such programs, and show that under these assumptions, hyperproperties reduce to trace properties and thus the monitoring problem reduces to monitoring of trace properties; firstly we assume that programs are deterministic, and secondly we assume that programs are executed repeatedly, but without persistent state between executions – in the manner of a stateless server. Under these assumptions we obtain a program in a loop, and a corresponding extension of the property of interest to traces. Specifically we make the following contributions:

i) We define a runtime monitor to check properties in Hyper_{2S}, parameterised by a predicate that characterises different properties in the class. The monitor operates over traces (and not over set of traces) due to a previous transformation of the monitored system by putting the program in a loop. We show that this transformation is sound (Section 3), and define a runtime verification monitor (Section 4). We provide a way to automatically synthesise the monitor. Based on the runtime verification monitor definition in Section 4, we present an online runtime verification algorithm for any property φ expressed in Hyper_{2S} (Section 5).
ii) We show that whenever the input domain is finite, using runtime monitoring in a controlled (pre-deployment) environment, gives a definite answer as to whether the program satisfies a given property φ or not. For the particular case of data minimisation, when monitoring in a controlled (pre-deployment) environment, we can extract a pre-processor (a *minimiser*) which may be composed with the system in order to guarantee minimality. In this case, we are thus synthesising an *enforcer* (Section 6).
iii) We implemented the above as a parameterised monitoring algorithm [2].

2 PRELIMINARIES AND NOTATIONS

A finite word over a finite alphabet Σ is a finite sequence $\sigma = a_1 \cdot a_2 \cdots a_n$ of elements of Σ. The set of all finite words over Σ is denoted by Σ^*. The *length* of a finite word σ is denoted by $|\sigma|$. The empty word over Σ is denoted by ϵ_Σ, or ϵ when clear from the context. The *concatenation* of two words σ and σ' is denoted as $\sigma \cdot \sigma'$. A word σ' is a *prefix* of a word σ, denoted as $\sigma' \preccurlyeq \sigma$, whenever there exists a word σ'' such that $\sigma = \sigma' \cdot \sigma''$; and $\sigma' \prec \sigma$ if additionally $\sigma' \neq \sigma$; conversely σ is said to be an *extension* of σ'.

[1]Termination insensitivity, or more generally *progress sensitivity* (see e.g. [4]), means that the property in question ignores the influence of progress or termination.
[2]https://github.com/SrinivasPinisetty/monDP_HyperP

Given a word σ of length n, for any $i \in [1, n]$, σ_i denotes i^{th} element in σ. Given two integers i and j where $1 \leq i \leq j \leq |\sigma|$, the *subword* from index i to j is noted $\sigma_{[i \cdots j]}$, and the suffix of word σ starting from index i is denoted as $\sigma_{[i \cdots]}$. Given an n-tuple of symbols $e = (e_1, \ldots, e_n)$, for $i \in [1, n]$, $\Pi_i(e)$ is the projection of e on its i-th element, i.e., $\Pi_i(e) \overset{\text{def}}{=} e_i$.

A trace property φ is a set of words over alphabet Σ, i.e., $\varphi \subseteq \Sigma^*$. Property φ is *prefix-closed* (also known as a safety property) if all prefixes of all words in φ also belong to φ (i.e., $\forall \sigma \in \varphi, \forall \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma \implies \sigma' \in \varphi$). Conversely, property φ is *extension-closed* (co-safety) if all extensions of words in φ also belong to φ (i.e., $\forall \sigma \in \varphi, \forall \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies \sigma' \in \varphi$).

2.1 Runtime verification monitor for trace properties

In this section, we present a definition of a monitor for any given trace property φ, and discuss some important properties that it satisfies.

A *runtime verification* (RV) monitor is a device that reads/observes a finite trace (an execution of the system being monitored) and emits a certain verdict regarding satisfaction of a given property φ. The verdicts provided by the monitor belong to the set $\mathcal{D} = \{\top, \bot, ?\}$, where verdicts true ($\top$), and false ($\bot$) are conclusive verdicts, and unknown (?) is an inconclusive verdict. A monitor for any given property φ is denoted as M_φ . Let us revise the definition of a verification monitor for any given trace property $\varphi \subseteq \Sigma^*$ [6].

Definition 2.1 (RV monitor). Let $\sigma \in \Sigma^*$ denote current observation of an execution of the system, and consider a property $\varphi \subseteq \Sigma^*$. A *runtime verification monitor* (RV monitor) is a function $M_\varphi : \Sigma^* \to \mathcal{D}$, where $\mathcal{D} = \{\top, \bot, ?\}$ defined as follows:

$$M_\varphi(\sigma) \quad = \begin{cases} \top & \text{if } \forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \varphi \\ \bot & \text{if } \forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \notin \varphi \\ ? & \text{Otherwise} \end{cases}$$

Property φ is a set of finite words over alphabet Σ (i.e., $\varphi \subseteq \Sigma^*$). Verdicts *true* ($\top$) and *false* ($\bot$) are conclusive verdicts, and verdict *unknown* (?) is an inconclusive verdict. $M_\varphi(\sigma)$ returns $\top$ if for any continuation $\sigma' \in \Sigma^*$, $\sigma \cdot \sigma'$ *satisfies* φ. $M_\varphi(\sigma)$ returns $\bot$ if for any continuation $\sigma' \in \Sigma^*$, $\sigma \cdot \sigma'$ *falsifies* φ. $M_\varphi(\sigma)$ returns unknown (?) otherwise.

Remark 1 (Monitorability). *A property $\varphi \subseteq \Sigma^*$ is* monitorable *[6, 15, 24] if for any observed word $\sigma \in \Sigma^*$, there exists a finite word $\sigma' \in \Sigma^*$ such that the property φ can be positively or negatively evaluated for $\sigma \cdot \sigma'$. That is, $\forall \sigma \in \Sigma^*, \exists \sigma' \in \Sigma^* : M_\varphi(\sigma \cdot \sigma') \in \{\top, \bot\}$.*

All safety (resp. co-safety) properties are monitorable [6, 15]. For a safety (resp. co-safety) property, a monitor can provide a conclusive verdict $\bot$ (resp. $\top$) when it observes a finite word that violates (resp. satisfies) the property. It is shown that safety and co-safety properties represent only a subset of monitorable properties [6, 14, 15]. Monitorable properties according to safety-progress classification of properties has been discussed in [15], where it is shown that Boolean combinations of safety and co-safety properties are monitorable. There are some response properties, such as "Every request is acknowledged", which are non-monitorable since for all finite words, it is never possible to decide satisfaction or violation of the property. This is because *every finite word can be extended both to a word that belongs to the property and to a word that does not belong to the property.*

Proposition 2.2. *For any given property $\varphi \subseteq \Sigma^*$ that is monitorable, monitor M_φ as per Definition 2.1 satisfies the following constraints:*

Impartiality $\forall \sigma \in \Sigma^*$,
$M_\varphi(\sigma) = ?$ *iff*
$$(\sigma \in \varphi \land \exists \sigma' \in \Sigma^* : \sigma \cdot \sigma' \notin \varphi) \lor (\sigma \notin \varphi \land \exists \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \varphi) \quad \textbf{(Imp)}$$

Anticipation $\forall \sigma \in \Sigma^*$,
$$\begin{array}{ll} M_\varphi(\sigma) = \top \ \textit{iff} \ (\forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \varphi) \\ M_\varphi(\sigma) = \bot \ \textit{iff} \ (\forall \sigma' \in \Sigma^* : \sigma \cdot \sigma' \notin \varphi) \end{array} \quad \textbf{(Acp)}$$

Impartiality expresses that for a finite trace $\sigma \in \Sigma^*$, the monitor provides inconclusive verdict ? if and only if there exists a continuation of σ leading to another verdict. That is, if σ is consistent with φ, but there is some extension of σ which is not, or conversely, if σ is not consistent with φ but some extension is, then the monitor must give verdict ? on σ.

Anticipation states that for a finite trace $\sigma \in \Sigma^*$, the monitor $M_\varphi(\sigma)$ should provide a conclusive verdict $\top$ (resp. $\bot$) iff every continuation of σ satisfies (resp. violates) φ. Thus, anticipation also means that if $M_\varphi(\sigma)$ is $\top$ (resp. $\bot$), then every continuation of σ also evaluates to $\top$ (resp. $\bot$).

Constraints **Imp** and **Acp** ensure that the monitor provides a conclusive verdict as soon as possible. The terms impartiality and anticipation are introduced as requirements of monitors in other works related to runtime verification for trace properties [18].

2.2 Deterministic programs

In this paper we consider monitoring deterministic programs which are executed repeatedly. In each execution of the program, it consumes an input, and emits an output.[3] Thus a single execution of such a program is an input-output event $(i, o) \in I \times O$, where I denote a finite set of inputs, and O denotes a finite set of outputs. Traces over $\Sigma = I \times O$ will be obtained by repeated execution. Since we focus on deterministic programs we will not work with Σ^*, but instead with $\Sigma^\#$, the subset of *deterministic traces*, given by the following property:

$$\forall \sigma \in \Sigma^\#, \forall i, j \in [1, |\sigma|], \text{ if } \Pi_1(\sigma_i) = \Pi_1(\sigma_j) \text{ then } \Pi_2(\sigma_i) = \Pi_2(\sigma_j).$$

$\Sigma^\#$ corresponds to traces obtained by making repeated runs of some deterministic program which has no history, and is thus the set of all finite words over alphabet Σ that do not contain input-output events which have the same input values but differ in their output values.[4]

We are interested in checking whether a program satisfies properties such as data minimality, integrity and software doping (introduced later in Section 2.4). These are in fact *hyperproperties* [11, 17], that is defined over sets of sets of traces. In order to monitor hyperproperties we need to consider multiple executions of the program being monitored and the analysis of sets of traces [8].

Since we consider the program being monitored to be executed repeatedly, as if in a loop, the properties we consider may be treated as trace properties instead of as hyperproperties, reducing the monitoring problem to the analysis of single traces (explained in Section 3). Note that this "transformation" is not as strange as it might seem. As an example, one may think of a program to be some service provided by a web server in such a way that its (service) functionality is not just used once but multiple times with (different) inputs (e.g., different clients invoking the service). In what follows we further elaborate on deterministic programs and their corresponding programs-in-loop.

[3]In the remainder of this paper, *deterministic programs* are referred to as *programs*.
[4]Σ^ω will denote all infinite words over alphabet Σ satisfying this determinism condition.

2.2.1 *Iterated deterministic programs.*

For certain properties it will be necessary to consider the inputs and outputs to be tuples. We consider a finite number of input sources $n \geq 1$, where the set of input events $I = I_1 \times \cdots \times I_n$. For all $id \in [1, n]$, I_{id} is a finite set of possible inputs for input source id, and an input event $(i_1, \cdots, i_n) \in I$, where $i_{id} \in I_i$. Similarly, we consider a finite number of outputs $m \geq 1$ the set of output events is $O = O_1 \times \cdots \times O_m$. In every execution of the program, it consumes an input event $(i_1, \cdots, i_n) \in I$, and emits an output event $(o_1, \cdots, o_m) \in O$. In order to describe security properties we classify each input (resp. output) as either *low* (*public*) or *high* (*secret*). For an input event $i \in I$, i_H denotes projection on high inputs, and i_L denotes projection on low inputs. Similarly, for an output event $o \in O$, o_H (resp. o_L) denote projection on high outputs (resp. low outputs).

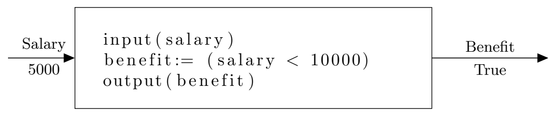
A program can be considered as a (partial) function denoted as $\mathcal{P} : I_1 \times \cdots \times I_n \to O_1 \times \cdots \times O_m$. The language of $\mathcal{P}$ is denoted as $\mathcal{L}(\mathcal{P}) = \{(i, o) \in I \times O : o = \mathcal{P}(i)\}$, and $\mathcal{L}(\mathcal{P}) \subseteq I \times O$. Note, $\forall i \in I, \forall o \in O, (i, o) \in \mathcal{L}(\mathcal{P}) \Rightarrow (\forall o' \neq o \in O : (i, o') \notin \mathcal{L}(\mathcal{P}))$. We monitor both inputs and outputs of a program $\mathcal{P}$. A single execution of $\mathcal{P}$ is an input-output event $(i, o) \in I \times O$.

As explained before, we consider observing (monitoring) the input-output behaviour over several executions of $\mathcal{P}$. Given a program $\mathcal{P}$, we denote by $\mathcal{P}l$ the transformed program consisting in putting $\mathcal{P}$ in a loop. We say that $\mathcal{P}l$ is the *program-in-loop* version of $\mathcal{P}$.[5] An execution of $\mathcal{P}l$ is an unbounded sequence of input-output events $\sigma \in \Sigma^\omega$, where $\Sigma = I \times O$.

The *behaviour* of program $\mathcal{P}l$ is denoted as $exec(\mathcal{P}l) \subseteq \Sigma^\omega$. The *language* of $\mathcal{P}l$ is denoted by $\mathcal{L}(\mathcal{P}l) = \{\sigma \in \Sigma^\#, \mid \exists \sigma' \in exec(\mathcal{P}l) \land \sigma \preccurlyeq \sigma'\}$ i.e. $\mathcal{L}(\mathcal{P}l)$ is the set of all finite prefixes of the sequences in $exec(\mathcal{P}l)$. Given a property $\varphi \subseteq \Sigma^\#$, we have that $\mathcal{P}l \models \varphi$ iff $\mathcal{L}(\mathcal{P}l) \subseteq \mathcal{L}(\varphi)$. Given a word $\sigma \in \Sigma^*$, $\sigma \models \varphi$ iff $\sigma \in \mathcal{L}(\varphi)$.

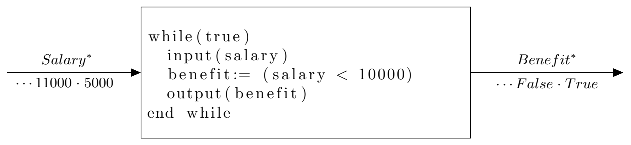
Example 2.3 (Program $\mathcal{P}$ and its corresponding program-in-loop $\mathcal{P}l$). Let us consider a simple example illustrated in Figure 1. In this example, the program has one input and one output. An example program $\mathcal{P} : I \to O$ is illustrated in Figure 1a, that takes salary information (which is an integer, i.e., set of possible inputs $I = \mathbb{N}$), and returns whether eligible for benefits or not (i.e., the set of possible outputs $O = \mathbb{B}$). The output of the function is true if salary is less than 10000, and false otherwise. Figure 1b illustrates an example of program $\mathcal{P}l$ which corresponds to repeated execution of $\mathcal{P}$ in Figure 1a. Thus, input to $\mathcal{P}l$ is a sequence of inputs events $\sigma_I \in I^*$, and the output is a of outputs $\sigma_O \in O^*$. In this example, $\sigma_I = 5000 \cdot 11000 \cdots$, and $\sigma_O = \mathsf{true} \cdot \mathsf{false} \cdots$. The set of input-output events $\Sigma = \mathbb{N} \times \mathbb{B}$, and a finite prefix of an execution of $\mathcal{P}l$ is $(5000, \mathsf{true}) \cdot (11000, \mathsf{false})$, where in the first iteration of the

[5]In the rest of the paper, $\mathcal{P}l$ will always denote the program-in-loop version of the program under consideration $\mathcal{P}$.



```
input(salary)
benefit:= (salary < 10000)
output(benefit)
```

(a) Example of a program $\mathcal{P}$



```
while(true)
  input(salary)
  benefit:= (salary < 10000)
  output(benefit)
end while
```

(b) Example of $\mathcal{P}l$ (repeated execution of $\mathcal{P}$)

Figure 1: A program $\mathcal{P}$ and its corresponding program $\mathcal{P}l$

while-loop, input is 5000 and output is true, and in the second iteration, input is 11000 and output is false. $(5000, \text{true}) \in \mathcal{L}(\mathcal{P}l)$, and $(5000, \text{true}) \cdot (11000, \text{false}) \in \mathcal{L}(\mathcal{P}l)$. □

2.3 Hyperproperties and Hyper$_{2S}$

Hyperproperties [11] are sets of trace properties, that can express complex security and privacy properties. Logics for verification of hyperproperties such as *HyperLTL* have been proposed [10]. In this section, we briefly revisit HyperLTL, and discuss the subset of HyperLTL we consider in our work. In Section 2.4 we will discuss some examples of security and information-flow properties and how they can be defined using the considered subset of HyperLTL.

Linear Temporal Logic (LTL) [23] implicitly quantifies over a single execution of a system. HyperLTL [10] generalises LTL allowing explicit quantification over multiple executions of a system, relating events occurrences in different traces. We refer readers to [9, 10] for details on syntax and semantics of HyperLTL.

Considered subset of HyperLTL. Note that HyperLTL formulas with alternating quantifiers are not monitorable in general, but this is not the case for the alternation-free fragment of HyperLTL [9]. The monitoring algorithms proposed for the alternation-free fragment are relatively complex since they focus on monitor synthesis for the set of all hyperproperties that are monitorable. Though the generality of such algorithms is highly appreciated, many security and information-flow properties commonly encountered in practice may be expressed as *2-safety* hyperproperties.

The concepts of *safety* and *liveness* for trace properties [2], are generalised to hyperproperties as *safety hyperproperties* and *liveness hyperproperties* [11]. Safety properties express that nothing bad should ever happen, where violation of a safety property is finitely observable. Thus, if a set of traces Tr violates a safety hyperproperty, this can be witnessed by a finite set $Tr' \subseteq Tr$ that will show an undesirable behaviour. If the size of Tr' never needs to be larger than k, then it is called a *k-safety hyperproperty*. For example, properties considered in Section 2.4 are *2-safety* hyperproperties since violation of those properties can be shown by observing two bad traces. Violation of a safety hyperproperty is irrecoverable, and thus if a set Tr violates a safety hyperproperty, then any extension of the set Tr also violates it. Formal characterisation of safety hyperproperties is given in [1].

We consider HyperLTL formulas without alternating quantifiers. Specifically, we restrict to 2-safety hyperproperties, expressed in HyperLTL in the following form $\forall \pi, \forall \pi' : \psi$ with:

$$\psi ::= a_\pi \mid \neg \psi \mid \psi \vee \psi$$

where π, π' are trace variables, and $a_\pi \in \Sigma$. Conjunction ($\wedge$), implication ($\rightarrow$) and bi-implication ($\leftrightarrow$) are defined using negation and disjunction.

A *co-safety hyperproperty* is a generalisation of the notion of co-safety for trace properties. Intuitively a co-safety property describes the occurrence of good things. Details about formal characterisation of co-safety hyperproperties are given in [1]. Note that a *2-co-safety* hyperproperty is of the form $\exists \pi, \exists \pi' : \psi$, which can be expressed as negation of a safety hyperproperty and vice-versa. We call this fragment Hyper$_{2S}$.

It is straightforward to extend the approach we present to formulas with more than two universal (resp. existential) trace quantifiers. We restrict here to two quantifiers only to ease presentation and since most interesting security and information-flow properties may be expressed using only two trace quantifiers.

2.4 Properties expressed in Hyper$_{2S}$

In this section, we consider some security and information-flow properties, and present how they can be formulated as safety hyperproperties using Hyper$_{2S}$.

Data minimisation. Data minimisation is a privacy enhancing principle, stating that personal data collected should be no more than necessary for the specific purpose consented by the user. Antignac et. al. [3] defined the concept of data minimiser as a pre-processor that filters the input of the given program in such a way that the functionality of the program does not change but it only receives data that is necessary and sufficient for the intended computation. From there they derived the concept of data minimisation and they showed how to obtain data minimisers for both the *monolithic* case (only one source of input) and the *distributed* case (more than one, independent, source of inputs).

We briefly discuss the data minimality principle in the monolithic case. The data minimality principle ensures that the range of inputs provided to a program is reduced such that when two inputs result in the same response, then one of them can be considered redundant. Ideally, a program satisfying the data minimisation principle should be one such that the cardinality of the output domain is equal to the cardinality of the input domain.

One way to achieve data minimality is to define an input pre-processor $Pre : I \longrightarrow I$, filtering the set of possible inputs I for the program $\mathcal{P}$ [3] . The idea of the function Pre is to transform the inputs before they are fed to program $\mathcal{P} : I \longrightarrow O$ as it it considered to be *un-trusted.*

Definition 2.4 (Pre-processor). Given program $\mathcal{P} : I \longrightarrow O$, we say that $Pre : I \longrightarrow I$ is a pre-processor for $\mathcal{P}$ iff:

(1) $\forall i \in I : \mathcal{P}(Pre(i)) = \mathcal{P}(i)$, and
(2) $\forall i \in I : Pre(i) = Pre(Pre(i))$.

```
inputPre(salary)
  if (salary < 6000)
    salaryRep = 1000
  else if  (6000 < salary <10000)
    salaryRep = 6000
  else:
    salaryRep= 10000
return(salaryRep)
```

Figure 2: An input pre-processor for $\mathcal{P}$ in Figure 1a.

Condition 1 states that the pre-processor should not change the behavior of the program. For any input $i \in I$, the output that the program produces by consuming the pre-processed input should be equal to the output it produces by directly consuming the input i. Condition 2 states that for any input $i \in I$, if we feed the pre-processed input to the pre-processor again, then it returns back the same pre-processed input.

Pre-processors perform some degree of domain reduction, and range(Pre) $\subseteq I$ (with range(Pre) denoting the range of function Pre, i.e., $\{Pre(i) | i \in I\}$. A trivial pre-processor is the identity function that simply forwards any user input to program $\mathcal{P}$.

Example 2.5 (Input data pre-processor). Figure 2 presents an example input pre-processor for program $\mathcal{P}$ illustrated in Figure 1a. If salary is less than 6000 it is mapped to representative 1000, if salary is greater than 6000 and less than 10000 it is mapped to 6000, and salary is mapped to 10000 otherwise.

Definition 2.6 expresses in Hyper$_{2S}$ that a program $\mathcal{P} : I \to O$ (with $I' \subseteq I$) is *monolithic minimal* for I' if for any two inputs $i_1, i_2 \in I'$, where i_1 is different from i_2, the output that program $\mathcal{P}$ produces for input i_2 should differ from the output that it produces for input i_1. Note that π and π' in the definition are traces of length 1 because of the sort of programs we consider in this work (see Section 2, Definition of $\mathcal{L}(\mathcal{P})$). For an input-output trace π, projection on inputs is denoted as π_I and projection on outputs is denoted as π_O.

Definition 2.6 (Monolithic minimality of program $\mathcal{P}$). Given program $\mathcal{P} : I \to O$, $\mathcal{P}$ is monolithic minimal for $I' \subseteq I$ *iff* the following property holds:

$$\forall\pi, \forall\pi' : ((\pi_I \in I' \land \pi'_I \in I') \land (\pi_I \neq \pi'_I)) \implies \pi_O \neq \pi'_O.$$

The Hyper$_{2S}$ formula in the above definition expresses that for any two traces π and π' where the inputs in both the traces belong to I' and are different (i.e., $\pi_I \neq \pi'_I$)), the outputs in both the traces should be also different (i.e., $\pi_O \neq \pi'_O$). So, for the monolithic case, data minimisation corresponds to *injectivity* (this is not the case for the distributed setting —see the accompanying appendix available online at [22]).

Note that when we say that $\mathcal{P}$ is (non-) minimal we mean that the composition of $\mathcal{P}$ with a pre-processor ($Pre : I \to I'$) is (non-) minimal. We consider the scenario illustrated in Figure 2. If composition of $\mathcal{P}$ with Pre is (non-) minimal, and if Pre is the identity function then $\mathcal{P}$ itself is (non-) minimal.

Property	Property expressed in Hyper$_{2S}$
Data minimisation (Monolithic minimality)	$\forall\pi, \forall\pi' : \pi_I \neq \pi'_I \implies \pi_O \neq \pi'_O.$
Non-Interference	$\forall\pi, \forall\pi' : (\pi_{I,L} = \pi'_{I,L}) \implies (\pi_{O,L} = \pi'_{O,L})$
Integrity	$\forall\pi, \forall\pi' : (\pi_{I,H} = \pi'_{I,H}) \implies (\pi_{O,H} = \pi'_{O,H})$
Software doping (doping free program)	$\forall\pi, \forall\pi' : ((\pi_{Parm} \in PIntrs) \land (\pi'_{Parm} \in PIntrs) \land (\pi_I = \pi'_I)) \implies (\pi_O = \pi'_O)$
Strong distributed minimality	$\forall\pi, \forall\pi',$ let $\pi_i = (i_1, \cdots, i_n), \pi'_i = (i'_1, \cdots, i'_n).$ $(\exists x \in [1, n] : i_x \neq i'_x \land \forall y \in [1, n] : y \neq x \implies i_y = i'_y) \implies \pi_o \neq \pi'_o.$

Table 1: Example of properties expressed in Hyper$_{2S}$

Other Properties. In this section (and the rest of the paper) we have restricted our presentation to data minimisation, and in particular to the monolithic case (programs with one input source).

The decision to restrict our presentation has been taken in order to simplify the explanation of the approach and for space considerations. In what follows we briefly describe other properties in the class we have considered, for which all our results apply. The properties are summarised in Table 1 and are further explained in the accompanying online appendix [22].

Distributed Minimality Our approach extends to the more complex case of *distributed* data minimisation (more than one independent input sources).

Noninterference A program satisfies *non-interference* [10] if the low (public) outputs only depend on the low inputs (i.e. are uninfluenced by the secrets). This can be expressed formally by saying that every pair of traces with the same low inputs have the same low outputs.

Integrity Integrity is just an alternative interpretation of non-interference [10]: traces having the same high inputs but possibly different low inputs should have the same high outputs (i.e. high integrity "trusted" outputs do not depend on low integrity "untrusted" inputs).

Doping A software system can be considered as *doped* if it includes behaviour that serves some hidden interest favouring a certain manufacturer, or vendor which cannot be justified by the interest of the licensee [5, 13]. We have considered one simple definition of characterisation of a *doping-free* software from [13] and formulated it using Hyper$_{2S}$.

3 FROM HYPERPROPERTIES TO TRACE PROPERTIES

Here we show how, for deterministic programs and Hyper$_{2S}$ properties, the monitoring problem can be reduced to the problem of monitoring trace properties.

When considering the program $\mathcal{P}$ to be executed in a loop, we reduce the problem of monitoring hyperproperties to monitoring trace properties. For the set of safety hyperproperties expressed in Hyper$_{2S}$, i.e., properties of the form $\forall\pi, \forall\pi' : \psi$, the transformation is straightforward. In the remainder of this paper we denote a hyperproperty expressed in Hyper$_{2S}$ as φ and its corresponding trace property as φ_T.

Definition 3.1 (φ_T corresponding to a given φ). Given a safety hyperproperty φ expressed in Hyper_{2S} (of the form $\forall\pi, \forall\pi' : \psi$), where $\pi, \pi' \in \Sigma$, its corresponding trace property φ_T is the set of all words $\sigma \in \Sigma^{\#}$ satisfying the following constraint: $\forall\sigma \in \varphi_T, \forall i \in [1, |\sigma|], \forall j \in [1, |\sigma|] \cdot \psi_T$, where ψ_T corresponds to the condition expressed by ψ in property φ by treating the element at index i as trace π and the element at index j as trace π', where both π and π' are traces of length one.

Example 3.2 (Data minimality as trace property). Consider φ to be the data minimality property discussed in Section 2.4: $\forall\pi, \forall\pi'$: $(\pi_I \neq \pi'_I) \implies \pi_O \neq \pi'_O$. Its corresponding trace property $\varphi_T \subseteq \Sigma^{\#}$, is the set of all words belonging to $\Sigma^{\#}$ satisfying the following constraint:

$$\forall\sigma \in \varphi_T, \forall i \in [1, |\sigma|], \forall j \in [1, |\sigma|], \\ \text{if } (\Pi_1(\sigma_i) \neq \Pi_1(\sigma_j)) \text{ then } (\Pi_2(\sigma_i) \neq \Pi_2(\sigma_j)).$$

Monolithic minimality, as trace property $\varphi_T \subseteq \Sigma^{\#}$, is the set of all words in $\Sigma^{\#}$, such that for any word $\sigma \in \varphi_T$, for any two events at different indexes in σ, if the projection on inputs of the two events differ, then the projection on outputs of the two events also differ.

Proposition 3.3 (φ_T is a (prefix-closed) safety property). *Given any safety hyperproperty φ expressed in Hyper_{2S}, its corresponding trace property φ_T is a (prefix-closed) safety property.*

Let $\sigma \in \Sigma^{\#}$ be a prefix of an execution of a program $\mathcal{P}l$. Given a hyperproperty φ, where φ_T is its corresponding trace property, we have that σ satisfies property φ_T iff $\sigma \in \varphi_T$.

Lemma 3.4 ($\overline{\varphi_T} = \Sigma^{\#} \setminus \varphi_T$). *Consider a given hyperproperty φ expressed in Hyper_{2S}, where φ_T is its corresponding trace property. The negation of property φ_T is denoted as $\overline{\varphi_T}$, where $\overline{\varphi_T} = \Sigma^{\#} \setminus \varphi_T$. A word $\sigma \in \Sigma^{\#}$ satisfies $\overline{\varphi_T}$ if $\sigma \in \overline{\varphi_T}$. We thus have that*

- $\forall\sigma \in \Sigma^{\#}, \sigma \in \varphi_T \implies \sigma \notin \overline{\varphi}_T$.
- $\forall\sigma \in \Sigma^{\#}, \sigma \in \overline{\varphi}_T \implies \sigma \notin \varphi_T$.

Let us consider $\mathcal{P}$ and its corresponding program-in-loop $\mathcal{P}l$. Consider a trace property φ_T (with $\overline{\varphi_T}$ its negation) corresponding to a hyperproperty φ. The following theorem states that, if there exists an observation of an execution of program $\mathcal{P}l$ that violates the property φ_T, then program $\mathcal{P}$ violates property φ. If every word that belongs to $\mathcal{L}(\mathcal{P}l)$ also belongs to property φ_T (i.e., every possible observation of execution of $\mathcal{P}l$ satisfies φ_T), then $\mathcal{P}$ satisfies property φ.

Theorem 3.5. *Given program $\mathcal{P}$, and property φ expressed in Hyper_{2S}, and the corresponding program-in-loop $\mathcal{P}l$ where $\mathcal{L}(\mathcal{P}l) \subseteq \Sigma^{\#}$, then the following hold:*

- *$\mathcal{P}$ violates property φ iff $(\exists\sigma \in \Sigma^{\#} : \sigma \in \mathcal{L}(\mathcal{P}l) \wedge \sigma \in \overline{\varphi}_T)$.*
- *$\mathcal{P}$ satisfies property φ iff $(\forall\sigma \in \Sigma^{\#} : \sigma \in \mathcal{L}(\mathcal{P}l) \Rightarrow \sigma \in \varphi_T)$.*

4 RUNTIME VERIFICATION MONITOR SYNTHESIS

Recall that an RV monitor for a trace property φ_T is a device that reads a finite trace and yields a certain verdict. We consider *online* verification monitoring, where the monitor treat executions in an incremental manner. Whenever the monitor observes a new event (thus changing the observed input-output word), it emits a verdict. A verdict is a value from the domain $\mathcal{D} = \{\top, \bot, ?\}$, where the verdict *unknown* (?) is an inconclusive verdict.

Let us recall that the set of input-output events that the monitor can observe (receive) as input is denoted using Σ, where $\Sigma = I \times O$, and an input-output event is denoted as (i, o) where $i \in I$ and $o \in O$. After n executions of the program $\mathcal{P}$, the monitor for a given property φ_T observes a word $\sigma = (i_1, o_1), \cdots, (i_n, o_n) \in \mathcal{L}(\mathcal{P})$ as input. φ_T denotes the trace property corresponding to a given property φ, and chk_φ is a predicate extracted from the property φ_T.

Definition 4.1 (chk_φ). Consider a property φ expressed in Hyper_{2S}, where φ_T denotes its corresponding trace property. We denote the predicate obtained from φ_T by ignoring the outermost universal quantifier (i.e., by ignoring $\forall\sigma \in \varphi_T$) as chk_φ.

Example 4.2 (Predicate chk_φ from property φ_T). Let the safety hyperproperty to be monitored φ be the monolithic minimality property in Definition 2.6. Its corresponding trace property φ_T is described in Example 3.2. The condition to check for (chk_φ) is obtained from φ_T by ignoring the outermost universal quantifier (i.e., by ignoring $\forall\sigma \in \varphi_T$). For the considered monolithic minimality example, chk_φ is: $\forall i \in [1, |\sigma|], \forall j \in [1, |\sigma|]$, if $(\Pi_1(\sigma_i) \neq \Pi_1(\sigma_j))$ then $(\Pi_2(\sigma_i) \neq \Pi_2(\sigma_j))$.

The following proposition expresses that if a word σ belongs to property $\overline{\varphi_T}$, then every possible extension of σ also belongs to $\overline{\varphi_T}$.

Proposition 4.3. $\forall\sigma \in \Sigma^{\#}$, *if* $\sigma \in \overline{\varphi_T}$ *then* $(\forall\sigma' \in \Sigma^{\#} : \sigma \preccurlyeq \sigma' \implies \sigma' \in \overline{\varphi_T})$.

Condition for conclusive verdict $\bot$. From Proposition 4.3, the condition of the second case of the monitor M_{φ_T} (Definition 2.1) can be reduced to checking whether the current observation σ satisfies $\overline{\varphi}_T$.

Remark 2 (Impossibility of checking condition of the first case (satisfaction of property φ_T)). *Note that regarding the condition of the $\top$ case (satisfaction of φ_T), checking whether the current observed word σ belongs to φ_T (i.e.,whether $chk_\varphi(\sigma)$ is* true*) is not sufficient, and does not ensure that every extension of σ will also belong to φ_T if σ belongs to φ_T. Thus, testing the condition of the first case is not possible in general.*

By providing the monitor with knowledge about the input domain and when the input domain is bounded, it is possible to test the condition of the first case related to satisfaction of property φ_T.

We introduce the function in-ex : $I \times \Sigma^{\#} \to \mathbb{B}$ in order to test whether every input belonging to I appears at least once in a given input-output word $\sigma \in \Sigma^{\#}$. It is defined as follows:

$$\text{in-ex}(I, \sigma) \quad = \begin{cases} \text{true} & \text{if } \ (\forall i \in I, \exists id \in [1, |\sigma|] : \Pi_1(\sigma_{id}) = i) \\ \text{false} & Otherwise \end{cases}$$

Condition for conclusive verdict $\top$. The condition of the first case in M_{φ_T} as per Definition 2.1 for property φ_T reduces to checking whether σ satisfies the following two conditions:

- every input belonging the set of inputs I appear at least once in σ, i.e., in-ex(I, σ) = true and
- σ satisfies φ_T, i.e., $chk_\varphi(\sigma)$ = true.

Note that if σ contains all possible inputs in I, and if σ satisfies φ_T, then every possible extension of σ also satisfies φ_T.

σ	$M(\sigma)$
$(5000, \text{true})$	?
$(5000, \text{true}) \cdot (11000, \text{false})$	?
$\mathbf{(5000, true)} \cdot (11000, \text{false}) \cdot \mathbf{(8000, true)}$	$\bot$
$\mathbf{(5000, true)} \cdot (11000, \text{false}) \cdot \mathbf{(8000, true)} \cdot (12000, \text{false}) \cdots$	$\bot$

Table 2: Example illustrating behavior of the monitor.

Proposition 4.4. *Given any word $\sigma \in \Sigma^{\#}$, where $\Sigma = I \times O$, and $|\sigma| > 1$, we have that if* $(\text{in-ex}(I, \sigma) \wedge chk_{\varphi}(\sigma))$ *then* $(\forall \sigma',$ *if* $\sigma \preccurlyeq \sigma'$ *then* $\sigma' \in \varphi_T)$.

Thus, using Propositions 4.3 and 4.4, the conditions of the first two cases in Definition 2.1 can be simplified. Consider property $\varphi_T \subseteq \Sigma^{\#}$. Recall that the RV monitor is a function $M_{\varphi_T} : \Sigma^{\#} \rightarrow \mathcal{D}$, where $D = \{\top, \bot, ?\}$. For $\sigma = \epsilon$ and any word σ of length 1, $M_{\varphi_T}(\sigma) =?$. Let $\sigma \in \Sigma^{\#}$ denote a finite input-output word over the alphabet Σ. M_{φ_T} is defined as follows:

Definition 4.5 (Monitor M_{φ_T}). Given property φ (where φ_T is its corresponding trace property), the RV monitor for property φ_T is a function $M_{\varphi_T} : \Sigma^{\#} \rightarrow \mathcal{D}$, where $D = \{\top, \bot, ?\}$, defined as follows:

$$M_{\varphi_T}(\sigma) \quad = \begin{cases} \top & \text{if } |\sigma| > 1 \wedge \text{in-ex}(I, \sigma) \wedge chk_{\varphi}(\sigma) \\ \bot & \text{if } |\sigma| > 1 \wedge \neg(chk_{\varphi}(\sigma)) \\ ? & Otherwise \end{cases}$$

with $\text{in-ex}(I, \sigma) = (\forall i \in I, \exists id \in [1, |\sigma|] : \Pi_1(\sigma_{id}) = i)$. chk_{φ} is a predicate extracted from φ_T.

For any word $\sigma \in \Sigma^{\#}$ (current observation of execution of $\mathcal{P}l$) where $|\sigma| > 1$, M_{φ_T}, as per Definition 2.1, is an RV monitor for property φ_T. The monitor returns $\top$ when σ followed by any extension of it satisfies the property φ_T. The monitor returns $\bot$ when the current observation of execution of $\mathcal{P}l$ followed by any extension of it violates φ_T (resp. satisfies $\overline{\varphi}_T$). It returns ? (unknown) for the current observation if the other two cases do not hold.

Proposition 4.6. *Given any property φ_T, M_{φ_T} (as defined in Definition 4.5) is a RV monitor for property φ_T (i.e., M_{φ_T} satisfies* **Imp** *and* **Acp***).*

Example 4.7 (Behaviour of the monitor M_{φ_T}). Let us again consider the monolithic minimality property introduced in Section 2.4 (cf. Definition 2.6; with its corresponding trace property φ_T as presented in Example 3.2). Let the program to be monitored be the one in Figure 1. In Table 2, we present some example observations of an execution of program $\mathcal{P}l$ being monitored, denoted as σ, and the verdict provided by the monitor for σ. Initially, when the first event observed in $(5000, \text{true})$, the monitor returns verdict unknown (?). In each step, the current observation is extended with a new event. Let the new event observed in the second step be $(11000, \text{false})$. For the current observation $\sigma = (5000, \text{true}) \cdot (11000, \text{false})$, the monitor returns verdict unknown. After observing the third event $(8000, \text{true})$, the monitor returns verdict false ($\bot$) for $\sigma = (5000, \text{true}) \cdot (11000, \text{false}) \cdot (8000, \text{true})$, and will return the same verdict for any extension of it. □

The following proposition states that changing the order of events does not effect the verdict given by the monitor.

Proposition 4.8 (Order of events does not matter). *Given φ (where φ_T correspond to its trace property),*

$$\forall \sigma \in \Sigma^{\#},\ \forall \sigma' \in \gamma(\sigma), (M_{\varphi_T}(\sigma) = \bot \implies M_{\varphi_T}(\sigma') = \bot) \wedge$$
$$(M_{\varphi_T}(\sigma) = \top \implies M_{\varphi_T}(\sigma') = \top).$$

where $\gamma(\sigma)$ are the set of all words obtained from σ by changing order. Each word in $\gamma(\sigma)$ is of same length as σ, and contains every element in σ (with possibly different index).

5 ONLINE MONITORING ALGORITHM

Algorithm 1 Monitor$_{\varphi}$

```
$\sigma \longleftarrow \epsilon$
$\varphi_T \longleftarrow$ get_trace_prop($\varphi$)
$C \longleftarrow$ get_condition($\varphi_T$)
while true do
    $(i, o) \longleftarrow$ await_event()
    if $\sigma = \epsilon$ then
        notify(?)
    else
        $eval \longleftarrow$ check_condition($C, \sigma \cdot (i, o)$)
        if $\neg eval$ then
            RETURN $\bot$
        else
            if in-ex($I, \sigma \cdot (i, o)$) then
                RETURN $\top$
            else
                notify(?)
            end if
        end if
    end if
    $\sigma \longleftarrow \sigma \cdot (i, o)$
end while
```

Consider property φ expressed in Hyper$_{2S}$, and a program-in-loop $\mathcal{P}l$ with input domain I. Algorithm 1 describes an online RV monitoring procedure for property φ.

In Algorithm 1, σ denotes the input-output word received by the monitor (i.e., observed execution of program $\mathcal{P}l$ being monitored). Function get_trace_prop takes property φ as input and returns its corresponding trace property, assigned to φ_T. Function get_condition takes a trace property as input and returns a predicate to be checked on the observed trace at runtime. Function check_condition takes a word σ and a predicate C and returns a Boolean, indicating whether σ satisfies C or not.

Algorithm 1 is an infinite loop that waits for input events, from the program $\mathcal{P}l$ being monitored. After each iteration, it receives an event (i, o) which is one execution of program $\mathcal{P}$. In the first iteration (i.e., when σ will be ϵ), verdict *unknown* is notified, and the algorithm proceeds to the next iteration. From the second iteration, after receiving the event (i, o), whether $\sigma \cdot (i, o)$ satisfies predicate C is checked using the function check_condition. If check_condition returns false, then the algorithm returns conclusive verdict $\bot$. Otherwise (when check_condition returns true), it is checked whether $\sigma \cdot (i, o)$ covers all the inputs in I using function in-ex. If in-ex$(I, \sigma \cdot (i, o))$ returns true, then the algorithm returns conclusive verdict $\top$. Otherwise (when check_condition$(C, \sigma \cdot (i,o))$ is true and in-ex$(I, \sigma \cdot (i,o))$ is false), the algorithm outputs ? (unknown), and proceeds to the next iteration. Before the next iteration, the observed trace is updated (i.e., event (i,o), is appended to the observed trace σ).

Proposition 5.1 ($|\sigma| < |I|$). *When the length of the input-output word $\sigma \in \Sigma^\#$ is less that the cardinality of the set of inputs, then* in-ex(I, σ) *is* false*:*
$\forall \sigma \in \Sigma^\# : |\sigma| < |I| \implies$ in-ex(I, σ) = false.

Note that when it is not possible to test whether the current observation σ covers all inputs (i.e., when the input domain I is unknown), it is not possible to compute in-ex(I, σ). In this case, one can consider a monitor with two cases (where the $\top$ case is merged with the unknown (?) case). The monitor returns $\perp$ indicating violation of the given property φ, and ? otherwise.

Remark 3 (Complexity of Algorithm 1). *The monitoring algorithm (Algorithm 1) has an offline and an online component. The offline component involves transformation to trace property φ_T and extracting the condition C to check at runtime. Both these operations have constant time complexity and thus the offline component has constant time complexity. Regarding the online component, the expensive computation is a check implemented by* check_condition(C, σ) *(i.e., test whether σ satisfies condition C). Note, in Algorithm 1, in some iteration k where (i,o) is the new event received,* check_condition$(C, \sigma \cdot (i,o))$ *in step 8 will be invoked only if* check_condition(C, σ) *(in the previous iteration) evaluates to* true. *If* check_condition(C, σ) *was* false *in step $k-1$, the algorithm would have returned $\perp$, and would not have entered iteration k. Thus in Algorithm 1, implementation of* check_condition$(C, \sigma \cdot (i,o))$, *can be simplified (since* check_condition(C, σ) *is* true*). Thus, to realise* check_condition$(C, \sigma \cdot (i,o))$, *the event received in the current step needs to be compared with every event in σ in the worst case, giving overall quadratic complexity in the length of the input trace.*

Implementation of Algorithm 1. Regarding implementabality of Algorithm 1, from a given property φ in Hyper$_{2S}$, getting its corresponding trace property φ_T (function get_trace_prop), and the condition to test on the observed execution at runtime (function get_condition) is straightforward as described in Section 3. The other functions used in Algorithm 1 are check_condition() and in_exh(), which are also straightforward to implement. Algorithm 1 has been implemented in Python that directly takes the condition to be checked (C) on the observed execution as input parameter. In order to validate the feasibility and practicality of the proposed approach, the implementation has been tested with some example properties such as data minimality and the other properties discussed in Section 2.4.

6 PRE-DEPLOYMENT TESTING AND ENFORCER SYNTHESIS

Consider a given safety hyperproperty φ, and its corresponding verification monitor as per Definition 4.5, given in Algorithm 1. In order to provide conclusive verdict $\top$ from an observed input-output word $\sigma \in \Sigma^\#$, testing whether σ satisfies (resp. violates) a condition C (extracted from φ) is not sufficient. In addition, the monitor has to be provided with information about the set of all possible inputs I, and it should check whether every possible input in I appears in σ at least once (i.e., test whether in-ex(I, σ) holds).

In runtime monitoring, the word $\sigma \in \Sigma^\#$ (observation of current execution of $\mathcal{P}$) that is fed to a monitor is of finite bounded length (σ and its length are both known). Thus, for any given $\sigma \in \Sigma^\#$ and any set of inputs I, testing in-ex(I, σ) is straightforward.

In general, when performing (online) monitoring of program $\mathcal{P}$ (where $\sigma \in \Sigma^\#$ is the current observation of execution of $\mathcal{P}$), and providing knowledge of the set of all possible inputs of $\mathcal{P}$ to the monitor, it is highly unlikely that σ covers all the inputs in I. Thus, during runtime monitoring, in-ex(I, σ) most likely will return false, and thus the condition of the first case (that provides conclusive verdict $\top$) in Definition 4.5 most likely does not hold, and thus we notice only verdicts ? or $\perp$ in practice.

However, for a given property φ, its monitor M_φ can be also used for testing for satisfaction (resp. violation) prior to deployment. In a testing environment, the input observation fed to the monitor can be generated in such a way that it covers all the inputs in I (when I is finite and bounded).

6.1 Testing in a controlled environment via monitoring

Algorithm 2 TestEnv

```
1: $I' \leftarrow I, \sigma \leftarrow \epsilon, v \leftarrow ?$
2: while $(|I'| > 0 \wedge v == ?)$ do
3:     $i \leftarrow pickInp(I')$
4:     $o \leftarrow \mathcal{P}(i)$
5:     $\sigma \leftarrow \sigma \cdot (i,o)$
6:     $v \leftarrow M_\varphi(\sigma)$
7:     $I' \leftarrow I' \setminus \{i\}$
8: end while
```

When the set of inputs I is bounded, and when testing $\mathcal{P}$ in a controlled environment (i.e., when we have control over the inputs that are fed to $\mathcal{P}$) for satisfaction (resp. violation) of a given safety hyperproperty, it is indeed possible to obtain a conclusive verdict (either $\top$ or $\perp$), upon observing a sequence σ of length $|I|$.

Algorithm 2 (TestEnv) is for testing $\mathcal{P}$ for satisfaction (resp. violation) of a given property φ via monitoring. In Algorithm 2, I' contains inputs that are not yet fed to the program $\mathcal{P}$. Initially, I' is assigned with the set of inputs I. In every iteration of the while loop, an input i from the set I' is picked non-deterministically, which is fed to $\mathcal{P}$, and $\mathcal{P}(i)$ is assigned to o. The input-output event (i,o) is then fed to the monitor. Before proceeding to the next iteration, input i which is already considered in the current iteration is removed from the set I'.

Algorithm 2 (TestEnv) can be considered as program $\mathcal{P}l$ where program $\mathcal{P}$ is executed repeatedly. However, here, in every iteration we invoke $\mathcal{P}$ with a new input from the set I (i.e., input that has not been considered in the previous iterations). The while loop thus terminates after $|I|$ iterations. After $|I|$ iterations of the algorithm, the input-output word σ that the monitor receives will be of length $|I|$, and in-ex(I, σ) will evaluate to true. The monitor M_φ returns a conclusive verdict upon receiving an input-output word of length $|I|$ (using Propositions 4.3 and 4.4).

Proposition 6.1. *Let $\sigma \in \Sigma^{\#}$ be an execution of the* TestEnv *(Algorithm 2), which is a sequence of input-output word fed to the monitor. The length of σ will be at most $|I|$, and the monitor will certainly return a conclusive verdict $\top$ or $\bot$ for σ of length $|I|$.*

Regarding the satisfaction of the property φ, the monitor *cannot* provide a verdict $\top$ before observing a word of length $|I|$. In what concerns the violation of the property φ, the monitor *may* provide a verdict $\bot$ before observing word of length $|I|$ (a witness of the violation may be found before exploring all the inputs). In the latter the execution of the tester program can stop earlier soon after the monitor observes the sequence that violates the property φ.

6.2 Enforcer synthesis

Runtime enforcement (RE) [16, 19, 25] is a technique to monitor the execution of a (black-box) system at runtime and ensure its compliance against a set of desired properties. The enforcer guarantees that the stream complies with a certain property, by delaying or modifying events if necessary.

For some hyperproperties expressed in Hyper$_{2S}$, it is possible to synthesise an enforcer for the property when the input domain I is bounded and known, and when we consider testing $\mathcal{P}$ via monitoring as discussed above. For example, when we consider the data minimality property introduced in Section 2.4, it is possible to synthesise an enforcer (input pre-processor), such that the composition of the enforcer with the program satisfies the data minimality property. For data-minimality we also implemented and tested the enforcer generation approach for some simple programs (see [20]). The synthesised enforcer (minimiser) acts as a input pre-processor of the program $\mathcal{P}$, that transforms inputs from the user (environment) before they are fed to the program.

7 RELATED WORK

The notion of *hyperproperties* has been proposed by Clarkson and Schneider [11] as a means to describe security policies that cannot be expressed as traditional trace properties. Hyperproperties generalize trace properties by also allowing to relate multiple traces, needed for example to express information-flow policies. HyperLTL and HyperCTL extend standard temporal logics LTL and CTL with explicit trace quantification enabling to express hyperproperties [10]. Algorithms for model checking HyperLTL and HyperCTL are given in [10, 17]. The complexity of the model checking of the alternation-free fragment of HyperLTL and HyperCTL formulae over finite-state kripke structures is shown to be PSPACE-complete in the size of the formula [10, 17].

Runtime verification monitoring mechanisms for trace properties has been extensively studied and several RV frameworks have been proposed such as [6, 7, 12, 15]. Most of the existing RV mechanisms are limited to the analysis of a single trace, and the RV problem of hyperproperties (dealing with sets of traces), is a recent and challenging problem. There are very few recent works dealing with runtime monitoring of HyperLTL. We focus here on [1, 9] as they are the most closely related to our work. We briefly discuss them in what follows and compare them with our approach.

In [1] an automata based approach for runtime verification of k-safety hyperproperties has been described. The algorithm in [1] can handle properties that reason/compare about position of events in different traces, which cannot be expressed in Hyper$_{2S}$. However, the algorithm in [1] is much more complex. The approach in [1] is based on runtime formula progression and *on-the-fly* monitor synthesis for LTL$_3$ sub-formulae across multiple executions, and computing runtime verdicts by aggregating progress of each sub-monitors. The complexity of the algorithm in [1] is mentioned as $O(\binom{n}{k} + \sum_{\phi \in \varphi} x^{\phi})$, where φ is given k-safety hyperproperty, n is the number of finite executions, and x^{ϕ} is the complexity of synthesising monitor for LTL sub-formula ϕ of φ.

In a later work in [9] a re-writing based monitoring approach for general alternation-free fragment of HyperLTL was proposed that has polynomial time and space complexity with respect to the number of traces. The approach in [9] involves extracting propositions of interest from the given HyperLTL formula, extraction of constraint c_i for each incoming trace t_i (involving re-writing on inner LTL formulae, and encoding what the monitor has observed). The current status/verdict is obtained by combining the evaluation of c_i for each individual trace.

Comparison. Similar to our work, the above discussed works for monitoring hyperproperties [1, 9] also consider the system being monitored to be a black-box. The set of hyperproperties we focus in our work (Hyper$_{2S}$) is clearly a subset of hyperproeprties considered in [1, 9]. However, most security and information-flow properties commonly encountered in practice for the considered setting (deterministic programs) can be expressed in Hyper$_{2S}$. As evident from the approaches for monitoring hyperproperties [1, 9], when we consider hyperproperties, RV mechanisms become very complex when compared with RV approaches for trace properties [6, 7, 12, 15]. The area/ideas of runtime verification (for trace properties) gained attention and interest (also in practice), due to its simplicity where the focus in on reasoning about a single execution trace. In this work, for the considered set of hyperproperties (Hyper$_{2S}$), we showed how the problem of monitoring Hyper$_{2S}$ properties can be reduced to the monitoring of trace properties, simplifying the monitoring algorithm. As discussed in Remark 3, an iteration of Algorithm 1 has linear complexity in the length of the trace. Additionally, we take advantage of the knowledge (assumption about determinism), and also integrate the possibility of providing conclusive verdict $\top$ for safety hyperproperties when the observed trace covers all possible inputs. General approaches for monitoring hyperproperties such as [1, 9] can never provide a conclusive verdict $\top$ for the considered safety hyperproperties also with knowledge about determinism, and when the input domain is finite and bounded.

8 CONCLUSION AND FUTURE WORK

In this paper, we considered the problem of runtime verification of safety hyperproperties for deterministic programs. Though there are a few runtime monitoring approaches for alternation-free fragment of HyperLTL, those algorithms are very complex compared to monitoring approaches for trace properties.

In this paper, for the considered subset Hyper$_{2S}$ and under the determinism assumption on the program being monitored, we reduced the problem to that of monitoring trace properties. We thus presented a simpler approach for runtime verification of

Hyper$_{2S}$ properties for deterministic programs. For any hyperproperty in Hyper$_{2S}$, we show how runtime verification monitors can be synthesised. Using the additional knowledge (assumptions about determinism and the input domain), we considered the possibility of providing conclusive verdict $\top$ when the observed trace covers all possible inputs. We also discussed about the possibility of synthesising monitors using the proposed approach in a pre-deployment testing phase.

Predictive runtime verification frameworks such as [21, 26] have been proposed for trace properties. In predictive monitoring frameworks [21, 26], instead of considering the system being monitored as a black-box, the monitor is provided with available knowledge about the system being monitored (i.e., the system is considered as a grey-box). The additional knowledge in some cases is useful for the monitor for instance to provide conclusive verdicts earlier when possible. The assumption we made about determinism can be seen as providing additional knowledge about the system being monitored. The problem of predictive monitoring for hyperproperties is a very important and unexplored one, and we intend to explore and generalise our results into that direction. We also intend to explore the problem of synthesising enforcers for the other properties in Hyper$_{2S}$, besides data minimisation.

Acknowledgements

This research has been partially supported by The Swedish Foundation for Strategic Research (SSF) under project DataBIN, the Swedish Research Council (*Vetenskapsrådet*) under grant Nr. 2015-04154 (*PolUser: Rich User-Controlled Privacy Policies*), and the European ICT COST Action IC1402 (*Runtime Verification beyond Monitoring (ARVI)*).

REFERENCES

[1] Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In *CSF*. 239–252.

[2] Bowen Alpern and Fred B. Schneider. 1984. *Defining Liveness*. Technical Report. Ithaca, NY, USA.

[3] Thibaud Antignac, David Sands, and Gerardo Schneider. 2017. Data Minimisation: A Language-Based Approach. In *IFIP SEC'17*, Vol. 502. 442–456.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. 2008. Termination Insensitive noninterference leaks more than just a bit. In *ESORICS*.

[5] Gilles Barthe, Pedro R. D'Argenio, Bernd Finkbeiner, and Holger Hermanns. 2016. Facets of Software Doping. In *ISoLA*. 601–608.

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4, Article 14 (Sept. 2011), 64 pages.

[7] Jan Olaf Blech, Yliès Falcone, and Klaus Becker. 2012. Towards Certified Runtime Verification. In *ICFEM*. Springer Berlin Heidelberg, 494–509.

[8] Borzoo Bonakdarpour and Bernd Finkbeiner. 2016. Runtime Verification for HyperLTL. In *RV*. Springer, 41–45.

[9] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. 2017. Rewriting-Based Runtime Verification for Alternation-Free HyperLTL. In *TACAS*. 77–93.

[10] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *POST*. 265–284.

[11] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210.

[12] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2009. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*. IEEE Computer Society, 33–37.

[13] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. 2017. Is Your Software on Dope?. In *ESOP*. Springer-Verlag New York, Inc., New York, NY, USA, 83–110.

[14] Volker Diekert and Martin Leucker. 2014. Topology, monitorable properties and runtime verification. *Theoretical Computer Science* 537, Supplement C (2014), 29 – 41. ICTAC 2011.

[15] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2009. *Runtime Verification of Safety-Progress Properties*. Springer Berlin Heidelberg, 40–59.

[16] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *FMSD* 38, 3 (2011), 223–262.

[17] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL. In *CAV*. Springer, 30–48.

[18] Martin Leucker. 2016. Runtime Verification for Linear-Time Temporal Logic. In *SETSS*. 151–194.

[19] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-Time Enforcement of Nonsafety Policies. *ACM Trans. Inf. Syst. Secur.* 12, 3, Article 19 (Jan. 2009), 19:1–19:41 pages.

[20] Srinivas Pinisetty, Thibaud Antignac, David Sands, and Gerardo Schneider. 2018. Monitoring Data Minimisation. *CoRR* abs/1801.02484 (2018).

[21] Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervè Marchand, and Viorel Preoteasa. 2017. Predictive runtime verification of timed properties. *Journal of Systems and Software* 132 (2017), 353 – 365.

[22] Srinivas Pinisetty, Gerardo Schneider, and David Sands. 2018. Appendix to the paper *Runtime Verification of Hyperproperties for Deterministic Programs*. https://github.com/SrinivasPinisetty/monitorDM/blob/master/Appendix.pdf. (Jan. 2018).

[23] Amir Pnueli. 1977. The Temporal Logic of Programs. In *SFCS '77*. IEEE Computer Society, Washington, DC, USA, 46–57.

[24] A. Pnueli and A. Zaks. 2006. *PSL Model Checking and Run-Time Verification Via Testers*. Springer Berlin Heidelberg, 573–586.

[25] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50.

[26] Xian Zhang, Martin Leucker, and Wei Dong. 2012. Runtime Verification with Predictive Semantics. In *NFM*. Springer Berlin Heidelberg, 418–432.