netidee

STIPENDIEN

Automated Diagnosis of Heisenbugs

Endbericht | Call 17 | Stipendium ID 6428

Inhalt

1 Einleitung

In unserer zunehmend digitalisierten Welt hängt die Gesellschaft in vielen Lebensbereichen von komplexen Computersystemen ab. Ob in der Industrie, im Gesundheitswesen oder in der Kommunikationstechnik – die Verlässlichkeit solcher Systeme ist entscheidend. Softwarefehler können erhebliche Risiken für Sicherheit und Zuverlässigkeit mit sich bringen und schwerwiegende finanzielle sowie gesellschaftliche Schäden nach sich ziehen.

Eine besonders komplexe Klasse von Softwarefehlern sind sogenannte *Heisenbugs*. Heisenbugs treten unregelmäßig auf und ändern ihr Verhalten, sobald man versucht, sie zu oder zu beheben. Der im Software Engineering gängige Begriff ist eine Anlehnung an das Heisenberg'sche Unschärfeprinzip.

Heisenbugs bringen besondere Herausforderungen sowohl für Nutzer:innen als auch für Entwickler:innen. Aus Nutzer:innen Sicht ist Folgende Situation ein klassisches Beispiel: Ein Programm läuft zehnmal hintereinander fehlerfrei, doch beim elften Versuch stürzt es plötzlich ab, scheinbar ohne, dass etwas an den Eingaben verändert wurde. Das ist problematisch, da das mit einem erlebten Kontrollverlust einher geht.

Aus Sicht von Entwickler:innen, ist die größte Herausforderung, dass Heisenbugs die Fehlersuche deutlich erschweren. Standardmäßig wird bei der Fehlersuche ein Debugger eingesetzt, um Programme Schritt für Schritt zu analysieren und Zwischenergebnisse zu überprüfen. Bei Heisenbugs kann der Debugger jedoch selbst das Verhalten des Programms beeinflussen, sodass der Fehler nicht mehr auftritt. Weiters führt die mangelnde Reproduzierbarkeit dazu, dass automatische Softwaretests, die dazu dienen, Fehler frühzeitig zu erkennen, versagen bei Heisenbugs oft, da der Fehler nur sporadisch oder unter bestimmten Bedingungen auftritt.

Der erste Schritt, um einen Heisenbug zu beheben ist es daher, herauszufinden, was die Ursache für die mangelnde Reproduzierbarkeit ist. Sobald man diese Ursache kennt, können oftmals Systemparameter gezielt kontrolliert werden, um den Fehler reproduzierbar zu machen und dann klassische Techniken zur Fehersuche einsetzen zu können.

Hier setzt meine Dissertation an. Das Ziel der Arbeit ist es, systematische Ansätze zu entwickeln, um Heisenbugs zu identifizieren, zu analysieren und ihre Ursachen zu verstehen.

Die entwickelten Ansätze bauen auf mathematischen Modellen und formalen Methoden auf. Formale Methoden haben den Vorteil, dass sie nicht nur einzelne Ausführungen eines Programms betrachten, sondern alle möglichen Szenarien systematisch untersuchen. Dadurch können selbst selten auftretende Fehler, wie Heisenbugs, identifiziert werden.

Das erse große Ergebnis meine Arbeit ist die Definition von Heisenbugs in Form einer *Hyperproperty*. Die mathematische Formalisierung dient zugleich als Grundlage für das zweite große Ergebnis: eine Methode zur kausalen Analyse, die aus einer Menge potentieller Ursachen feststellt, was tatsächlich die Ursache für das Auftreten eines Heisenbugs ist. Um leichter potentielle Ursachen sammeln zu können, habe ich eine systematsiche Taxonomie von häufigen Ursachen erstellt.

Sämtliche formale Methoden zur Fehlersuche und insbesondere zur Suche nach Heisenbugs basieren auf formalen Spezifikationen, die es erlauben korrekte von fehlerhaften Ausführungen zu unterscheiden. Wie sich im Laufe meiner Arbeit herausgestellt hat, ist es jedoch in der Praxis oft eine große Herausforderung passende formale Spezifikationen zu formulieren. Aus dieser Problematik, hat sich ein zweites Teilprojekt entwickelt, das ein wichtiger Teil meiner Dissertation ist. In diesem Teilprojekt habe ich eine Methode namens *Differential Property Monitoring* entwickelt, die bei der iterativen Verbesserung von Spezifikationen unterstützt. Die entwickelte Methode ist gut geeignet Spezifikationen für die Suche nach Heisenbugs zu entwickeln, ist aber auch allgemeiner für andere Problemstellungen anwendbar, die auf formalen Methoden basieren.

Meine Ergebnisse zeigen nicht nur neue Wege für die Erforschung von Heisenbugs auf, sondern weisen auch auf vielversprechende Möglichkeiten hin, basierend auf formalen Methoden praktische Werkzeuge für den Einsatz in der Softwareentwicklung zu entwickeln.

2 Allgemeines

Die zentrale Fragestellung meiner Arbeit lautet:

Wie können Heisenbugs formalisiert und ihre Ursachen automatisch identifiziert werden?

Diese Frage wird anhand mehrerer Teilaspekte untersucht:

Der erste Schwerpunkt liegt auf der Frage, wie man mathematisch präzise beschreiben kann, was ein Heisenbug ist. Eine solche formale Definition stellt die Grundlage für jegliche Formale Analysen zur Diagnose von Heisenbugs dar. In meiner Arbeit präsentiere ich ein Modell, das Heisenbugs anhand von einer Hyperproperty beschreibt. Hyperproperties beschreiben Beziehungen zwischen mehreren Programmausführungen. Die Definition besagt, dass ein Heisenbug vorliegt, wenn es in einem Programm zwei Ausführungen mit gleichen Nutzereingaben gibt, sodass eine Ausführung korrekt und eine Ausführung fehlerhaft ist.

Der zweite Aspekt widmet sich den Fragen, welche Quellen von Nichtdeterminismus im System Heisenbugs auslösen können und wie man kausale Zusammenhänge zwischen diesen Quellen und dem Auftreten von Heisenbugs analysieren kann. Nichtdeterminismus liegt vor, wenn das Systemverhalten nicht eindeutig durch die Nutzereingaben festgelegt ist. In meiner Arbeit präsentiere ich eine Taxonomie von häufigen Quellen nichtdeterministischen Verhaltens. Weiters habe ich ein Framework zur Kausalitätsanalyse entwickelt, das auf der Hyperproperty in der formalen Definition aufbaut. Dazu wird das Modell erweitert, indem Quellen von Nichtdeterminismus explizit als zusätzliche Eingaben dargestellt werden. Die Analyse basiert dann auf der Auswertung einer erweiterten Hyperproerty, die zwei Ausführungen sucht, die zwar zu unterschiedlichen Ergebnissen führen, aber in möglichst vielen Eingaben, die nichtdeterministische Mechanismen modellieren, übereinstimmen. Ursache für den Heisenbug sind dann all jene nichtdeterministischen Mechanismen, in denen sich die Ausführungen weiterhin unterscheiden.

Ein weiterer Aspekt ist die Frage, wie gute Spezifikationen für die Analyse und Diagnose von Heisenbugs entwickelt werden können. Präzise und korrekte formale Spezifikationen sind essenziell, um formale Methoden zur Fehlererkennung und -diagnose wirksam einzusetzen. Meine Arbeit stellt die Methode *Differential Property Monitoring* vor, die dabei hilft, Diskrepanzen zwischen verschiedenen Versionen von Spezifikationen zu identifizieren. So können Spezifikationen iterativ verbessert werden und

Fehler in der Spezifikation vermieden werden, die andernfalls die Diagnose verfälschen könnten.

3 Ergebnisse

Während der Laufzeit des Stipendiums wurden bisher drei wissenschaftliche Publikationen erstellt, wobei eine davon noch unter Peer Review ist. Die beiden bereits veröffentlichten Papers sind im Anhang zu diesem Bericht zu finden und stehen auf der Projekt Homepage zum Download bereit. Die Veröffentlichungen befassen sich mit unterschiedlichen Teilaspekten der oben genannten Ziele:

(1) Sallinger, S., Weissenbacher, G., Zuleger, F. (2023). A Formalization of Heisenbugs and Their Causes. In: Ferreira, C., Willemse, T.A.C. (eds) Software Engineering and Formal Methods. SEFM 2023. Lecture Notes in Computer Science, vol 14323. Springer, Cham. https://doi.org/10.1007/978-3-031-47115-5_16

(2) Sallinger, S., Weissenbacher, G., Zuleger, F. A Formalization of Heisenbugs and Their Causes (Extended Version) (erweiterte Journal Version von (1), unter Review beim International Journal on Software and Systems Modeling (SoSyM))

(3) Brechelmacher, O., Ničković, D., Nießen, T., Sallinger, S., Weissenbacher, G. (2024). Differential Property Monitoring for Backdoor Detection. In: Ogata, K., Mery, D., Sun, M., Liu, S. (eds) Formal Methods and Software Engineering. ICFEM 2024. Lecture Notes in Computer Science, vol 15394. Springer, Singapore. https://doi.org/10.1007/978-981-96-0617-7_13

(1) beinhaltet die formale Definition von Heisenbugs, sowie das Framework zur Kausalitätsanalyse wie oben beschrieben.

(2) ist eine erweiterte Journal Version von (1). Einzelne Papers der SEFM 2023 wurden eingeladen eine solche erweiterte Version für eine Veröffentlichung im International Journal on Software and Systems Modeling (SoSyM) zu erstellen. Die erweiterte Version enthält die genaue Taxonomie von Quellen von Nichtdeterminismus. Weiters erörtert sie, wie

man die Kausalitätsanalyse in der Praxis basierend auf Software Tests umsetzten kann.

(3) stellt die Methode des Differential Property Monitoring vor und präsentiert, wie die Methdode angewandt werden kann, um Spezifikationen für die Suche Backdoors in sicherheitskritischen Systemen zu erstellen. Backdoors sind Hintertüren, die von Herstellern und Entwicklern absichtlich in Systeme eingebaut werden, um später unerlaubterweise auf Systeme zugreifen zu können. Spezifikationen in diesem Kontext zu entwickeln ist besonders herausfordernd, da eine gute Spezifikation dem möglichen Angreifer immer einen Schritt voraus sein muss. Ein Highlight der Publikation ist der Nachweis, dass die Methode geeignet ist, eine passende Spezifikation für die Suche nach einer Backdoor in der Linux XZ-Bibliothek zu entwickeln.

4 Geplante weiterführende Aktivitäten

Im letzten Teil meiner Arbeit befasse ich mich zur Zeit mit konsistenzbasierter Diagnose für die Fehlerlokalisierung in Software. Die Grundidee dieser Methode ist es, formale Methoden zu nützen um im Falle eines Software Fehlers auf Programmzeilen hinzuweisen die mögliche Ursachen für den Fehler sind. Das stellt eine komplimentäre, ergänzende Methode zur kausalen Analyse von Heisenbugs dar.

Diese Arbeit ist in Zusammenarbeit mit einem Masterstudenten entstanden, dessen Masterarbeit ich mitbetreut habe. Die Masterarbeit ist im TU reposiTUm verfügbar:

> Graussam, L. (2024). *Consistency-based Software Fault Localization with Multiple Observations* [Diploma Thesis, Technische Universität Wien]. reposiTUm. https://doi.org/10.34726/hss.2024.109000

Der Fokus der Arbeit ist Fehlerlokalisierung basierend auf mehreren fehlerhaften Ausführungen. Im Zuge unserer Zusammenarbeit, haben wir zusätzlich an einem verfeinerten Fehermodell gearbeitet und untersucht, wie sich Optimierungen, die in bestehenden Ansätzen häufig sind, auf die Präzision der Feherlokalisierung auswirken. Zur Zeit arbeite ich daran diese Ergebnisse zu verfeinern und eine wissenschaftliche Publikation dazu zu verfassen. Der momentane Zeitplan ist es diese Publikation Ende April einzureichen.

Wie im Zwischenbericht und Planungsdokument erwähnt, ist ein weiterer und bisher unvollendeter Schritt meiner Arbeit die Veröffentlichung meiner Arbeit zum Thema „Memoization Bugs in JavaScript Programmen“, die eine spezielle Art von Heisenbugs darstellen. Bisher wurde die dafür erstellt Pulikation nicht zur Veröffentlichung akzeptiert. Ich plane das Paper in der vorliegenden Fassung neu einzureichen und eine Überarbeitung vorzunehmen, falls es der Zeitplan vor dem Abschluss meines Doktorats zulässt.

Ab Mai plane ich meine Dissertation zu verfassen, die die vorliegenden Ergebnisse in einen umfassenden wissenschaftlichen Rahmen einbettet und durch eine ausführliche Diskussion in den Kontext der bestehenden Literatur stellt.

5 Anregungen für Weiterführung durch Dritte

Meine Dissertation bietet einige vielversprechende Ansatzpunkte für weiterführende Forschungs- und Entwicklungsprojekte.

Ein wichtiger Bereich ist die praktische Umsetzung und Anwendung der Ansätze zur Diagnose von Heisenbugs. Die in der Dissertation vorgestellten theoretischen Grundlagen, insbesondere die Formalisierung von Heisenbugs und das Kausalitätsframework, legen eine fundierte Basis für die Entwicklung von praktischen Analysetools. So wäre es zum Beispiel denkbar existierende Debugging Tools mit eine Analysekomponente für Heisenbugs und deren Ursachen zu erweitern. Eine Integration in existierende Tools wie sie zum Beispiel in IDEs und Continuous Integration Systemen üblich sind, hätte den Vorteil, dass die Analysen für Entwickler:innen leichter zugänglich sind.

Ein zweiter Aspekt, bei dem eine Integration in bestehende praktische Arbeitsprozesse interessant wäre, ist die Methode des Differential Property Monitoring. Die Methode bietet Hilfestellung für die Suche nach präszisen formalen Spezifikationen. Dieses Problem ist relevant überall wo formale Methoden eingesetzt werden. Insbesondere wäre es auch spannend zu sehen, welchen Mehrwert die Methode in anderen Anwendungsbereichen außer der Suche nach Backdoors bringt.

Sobald ausgereifte Tool-Implementierungen vorliegen, wäre es aus wissenschaftlicher Sicht für beide genannten Bereiche spannend User Studies durchzuführen, um die Effektivität der Analysen in der Praxis zu testen.

A Formalization of Heisenbugs and Their Causes

Sarah Sallinger, Georg Weissenbacher, and Florian Zuleger

TU Wien, Vienna, Austria
firstname.lastname@tuwien.ac.at

Abstract. The already challenging task of identifying the cause of a bug becomes even more cumbersome if those bugs disappear or change their behavior under observation. Such bugs occur in a range of contexts including elusive concurrency bugs as well as unintended system alterations during debugging and—as a pun on the name of Werner Heisenberg—are often referred to as Heisenbugs. Heisenbugs can be caused by various sources of nondeterminism on different system levels, many of which developers and testers might not even be aware of. This paper provides formal foundations for rigorously reasoning about causes of Heisenbugs. It provides a formal definition of Heisenbugs in terms of a hyperproperty and introduces a framework for determining the causality of Heisenbugs in presence of multiple candidate causes based on said hyperproperty. We analyze the properties of causes and the implications on practical causal analyses.

1 Introduction

Bugs which change their behavior under observation are notoriously difficult to detect and fix. Inspired by Heisenberg's uncertainty principle such bugs are often referred to as *Heisenbugs*. Depending on the context, the term Heisenbug has been used to describe slightly different concepts. In the software engineering community, the term is used mostly for bugs whose analysis is hampered by the probe effect, i.e., an unintended alteration of the system behavior during debugging [18]. In the formal methods community, the term has been used to refer to elusive faults arising from executions that exhibit nondeterminism, in particular in the context of concurrent software [30]. In the context of automated testing, the term flaky test is used for inconsistently failing test cases [31], i.e. manifestations of Heisenbugs. As will become apparent in this paper, all the mentioned phenomena can be formalized in a uniform manner. In the rest of the paper, we hence use the term Heisenbug to refer to all the mentioned categories[1].

A Formalization of Heisenbugs. The first contribution of this paper is a formal definition of Heisenbugs. The unifying characteristic of Heisenbugs in the above-mentioned categories is the existence of at least two system executions where one

[1] In the literature, sometimes the term Mandelbug is used as an umbrella term for the mentioned categories. However, Mandelbugs additionally include complex faults where there is "a delay between the fault activation and the final failure occurrence".

execution is correct and the other exhibits a bug. In terms of testing, the same test case sometimes succeeds and sometimes fails. We formalize this definition in terms of a hyperproperty [7], which checks for the existence of two terminating executions with equal inputs but deviating outcomes for a final assertion that is part of the system specification. Our definition accommodates deviations caused by nondeterminism in a single program, e.g. due to concurrency, as well as deviating behavior of different versions of the program, e.g. due to changes for debugging.

Debugging Challenges. Previous studies have shown that Heisenbugs are prevalent even in mature software systems and that the bug fixing process takes significantly longer than for ordinary bugs [9]. Furthermore, Heisenbugs significantly complicate automated testing techniques, as they lead to flaky tests [31].

A major step in the debugging process is the identification of the bug's root causes [35]. Developers reported this step to be particularly difficult for Heisenbugs [11] (referred to as flaky tests in this study). One reason for the complexity is that Heisenbugs can be caused by *mechanisms* (i.e., sources of nondeterminism or system alterations) located on all system levels ranging from the hardware level to the user program. The following examples illustrate some possible causes:

Example 1 (Concurrency). We first present an example for a Heisenbug stemming from system internal nondeterminism. The Therac-25 incident [24,37], which resulted in the death of several cancer patients, is a notorious instance of an atomicity violation [27]. Listing 1.1 illustrates the problem, which is caused by the concurrent execution of two routines: the `userInterface` routine allows the operator to choose between high energy x-ray therapy (`isXray`) and a lower energy electron beam therapy (`!isXray`) and to set the intensity of the radiation (`isHigh`). The assume statement in line 6 prevents a selection of high-intensity electron therapy. The `setup` routine then processes these inputs: a failed assertion represents the case where the patient is exposed to excessive radiation. Assume that the user changes the initial configuration from high-intensity x-ray treatment (`isXray=true`, `isHigh=true`) to low-energy electron therapy (`isXray=false`, `isHigh=false`) in lines 4 and 7. If a context switch occurs right after executing line 5, the assertion in line 13 will fail. When `userInterface` is executed atomically, however, the assertion always holds.

```
1  bool isXray = true;
2  bool isHigh = true;
3  void *userInterface(void *a) {
4    isXray = read();
5    bool isHighTmp = read();
6    assume(isXray || !isHighTmp);
7    isHigh = isHighTmp;
8  }
9  void *setup(void *a) {
10   bool filter = isXray;
11   bool highEnergy = isHigh;
12 }
13 assert(filter || !highEnergy);
```

Listing 1.1: Illustration of Therac-25 atomicity violation

To sum up, there is a Heisenbug caused by different possible schedules, which is an example for the category of Heisenbugs arising from nondeterministic systems.

Even if the scheduler might in fact be deterministic, its internal steps are not observable for the programmer (which we model using nondeterminism).

Example 2 (Floating Point Precision). A prominent example for unintented system alterations are debugging statements that inadvertently change program outcomes. Consider Listing 1.2 (following [28]), which computes the square of 10^{308} and is expected to cause an overflow given the double-precision floating-point representation. When compiled with optimization level `-O3` and executed using x87 instructions, however, the computation results in 10^{308} rather than in an overflow, and the assertion fails. The reason is that the computation uses 80-bit floating point registers and performs rounding only once values are stored in 64-bit memory cells. Adding the `printf` statement in line 4 enforces such a write to memory, thus yielding the expected overflow, and the assertion holds.

```
1 double v = 1E308;
2 double y = 0;
3 y = v * v;
4 // printf("%g\n", y);
5 assert(isinf(y / v));
```

Listing 1.2: Floating-point computation overflows in case of `printf`-debugging

The failing and correct executions actually stem from two different system versions. In the considered execution model, the debugging statement changes the semantics of the program, introducing a probe effect which causes the Heisenbug.

Multiple Causes. While the mechanisms causing the Heisenbugs in Example 1 and Example 2 can still be easily identified, such an analysis becomes more challenging for more complex systems where several such mechanisms interact in a non-trivial manner [33] (as in Example 3 below).

Example 3 (Weak Memory Models). Listing 1.3 shows Peterson's mutual exclusion algorithm for two processes. Computer architectures with weak memory models relax the guarantees on the order in which variable assignments are observed across processor cores, causing the algorithm to fail. In particular, the synchronization fails if both processes set their flags in lines 5 and 15 but do not commit the modifications from cache to shared memory before lines 8 and 18 are executed, thus resulting in a Heisenbug (see [34]). Such a reordering, however, is effectively prevented if there are `printf` statements in lines 7 and 17 (as might be the case during development) and hence the bug only occurs once the `printf` statements are removed. Yet, the `printf` statements are not causally related to the Heisenbug (unlike in Example 2), as we will formally argue in Section 3.

Formal Causality Framework. In order to rigorously determine which mechanisms cause a Heisenbug in settings with multiple candidate causes, we present a formal causality defintion based on Lewis' counterfactuals [26] and the causality framework of Galles and Pearl [14]. In counterfactual reasoning, an event is a cause of an effect, if in an alternative world where the cause does not occur, the effect does also not occur. In a nutshell, in a setting with multiple candidate mechanisms, a subset of the mechanisms is a cause of a Heisenbug if there are

```
int flagP0 = 0, flagP1 = 0;
int turn = 0;
int critical = 0, error = 0;
void *petersonP0(void *a) {
  flagP0 = 1;
  turn = 1;
  //printf("barrier");
  while (flagP1 && (turn == 1));
  critical++;
  if (critical != 1) error++;
  critical--;
  flagP0 = 0;
}
```

```
void *petersonP1(void *a) {
  flagP1 = 1;
  turn = 0;
  //printf("barrier");
  while (flagP0 && (turn == 0));
  critical++;
  if (critical != 1) error++;
  critical--;
  flagP1 = 0;
}
assert(error == 0);
```

Listing 1.3: Peterson's algorithm occasionally fails on weak memory models

correct as well as failing executions which agree on the behavior of all other given mechanisms. This is formalized by means of a hyperproperty resembling our formal definition of Heisenbugs.

Note that our formal definition of causes refers to alternative scenarios for counterfactual reasoning. This requires the sources of nondeterminism to be made explicit in the underlying model (or controllable in the system under test, respectively). In practice, however, identifying and controlling all possible sources of nondeterminism is hardly feasible. Therefore, we prove that our causal analysis yields sound results even if some sources of nondeterminism remain unknown or uncontrollable: the result of evaluating our causality hyperproperty in a nondeterministic system is always a subset of a cause identified in the corresponding determinized system in which all sources are made explicit and controllable.

Based on these results, we present an iterative refinement methodology for causal analysis and discuss practical challenges. We showcase how the methodology can be applied for analyses based on model checking and testing.

Main Contributions. The paper presents:

- A formal definition of Heisenbugs in reactive systems in terms of a hyperproperty, in presence of system-internal nondeterminism and/or unintended system alternations (Section 2).
- A hyperproperty-based approach for defining the causality of Heisenbugs in the presence of several potential causes and nondeterminism (Section 3).
- A methodology for causal analysis based on iterative refinement (Section 4).

2 A Formalization of Heisenbugs

This section provides our system model and a formal definition of Heisenbugs.

2.1 System Model

In the following, the term *formula* refers to a first-order formula with a background theory that fixes the interpretations of predicates and function symbols.

Definition 1. *A* Symbolic Transition System *(STS) is a tuple* $(X, I, \mathsf{init}, \mathsf{final}, T)$*, where* X *and* I *are disjoint sets of* system *and* input variables*, respectively, the* initial condition init *is a formula over* $X \cup I$*, the* final condition final *is a formula over* X*, and the* transition relation T *is a formula over* $X \cup I \cup X'$*, where the variables* X' *denote primed copies of the variables* X*.*

Let Val *be a domain of* values*. Following [38], we assume* Val *to contain a special value* τ *that represents* quiescence*, i.e., the absence of an input. A* configuration c *of an STS is a mapping of the variables in* $(X \cup I)$ *to values in* Val*. A* state s *is a mapping of the variables in* X *to values in* Val*. An* input i *is a mapping of the variables in* I *to values in* Val*. The state* $c|_X$ *resp. input* $c|_I$ *of a configuration* c *is the restriction of the mapping to variables in* X *resp.* I*. We write* $c(v)$ *for the value of a variable* $v \in (X \cup I)$ *in configuration* c *(and we use the same notation for states and inputs).*

For a formula φ *and a mapping* m *of variables to values we write* $m \models \varphi$ *if* φ *evaluates to true under* m*. A configuration* c *is* initial *if* $c \models \mathsf{init}$*. A state* s *is* final *if* $s \models \mathsf{final}$*. A configuration* c *is* final *if* $c|_X$ *is final. A state* $s : X \to \mathsf{Val}$ *is* successor state *of configuration* c *if* $\langle c, s' \rangle \models T$*, where* $s' : X' \to \mathsf{Val}$ *is the function that maps each primed variable* $x' \in X'$ *to* $s(x)$ *and* $\langle \cdot, \cdot \rangle$ *denotes the union of two mappings with disjoint domains. We call* c_{i+1} *a* successor configuration *of* c_i *if* $c_{i+1}|_X$ *is a successor state of* c_i*. We require that final configurations do not have successor configurations.*

A (finite or infinite) trace *of an STS is a sequence of configurations* $c_0, \ldots, c_n$ *where* $c_0 \models \mathsf{init}$*, and* c_{i+1} *is a successor of* c_i *for all* $i \geq 0$*. An* execution *of an STS is a finite trace* $c_0, \ldots, c_n$ *such that* c_n *is a final configuration.*

It is straightforward to represent programs such as the examples from the introduction as symbolic transition systems:

Example 4. Listing 1.1 can be modeled as an STS with $I = \{\mathsf{input}_1, \mathsf{input}_2\}$ and $X = \{\mathsf{isXray}, \mathsf{isHigh}, \mathsf{isHighTmp}, \mathsf{filter}, \mathsf{highEnergy}, \mathsf{pc}_0, \mathsf{pc}_1\}$, where the variables pc_0 and pc_1 model the program counters of the two threads. The initial condition is $(\mathsf{isXray} \wedge \mathsf{isHigh} \wedge \mathsf{pc}_0 = 4 \wedge \mathsf{pc}_1 = 10)$. The final condition is $(\mathsf{pc}_0 = 8 \wedge \mathsf{pc}_1 = 12)$ and describes that both traces have reached their final program location. The transition relation T shown in Figure 1 is a disjunctive partitioning that represents a case split over all possible combinations of program locations, where the thread to be executed in each step is chosen nondeterministically.

While Example 4 illustrates the case of nondeterminism in a single system version, we next exemplify how to model system alterations in our formal model: the original and the altered system can be combined in one STS with an initial nondeterministic choice between two disjuncts of the transition relation.

$$
\begin{array}{lll}
 & \overbrace{\text{control flow}} & \overbrace{\text{data flow}} \\
 & (\mathsf{pc}_0 = 4 \wedge \mathsf{pc}_0' = 5) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isXray}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{isXray}' = \mathsf{input}_1) \\
\vee & (\mathsf{pc}_0 = 5 \wedge \mathsf{pc}_0' = 7) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isHighTmp}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & (\mathsf{isHighTmp}' = \mathsf{input}_2) \wedge (\mathsf{isXray}' \vee \neg \mathsf{isHighTmp}') \\
\vee & (\mathsf{pc}_0 = 7 \wedge \mathsf{pc}_0' = 8) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge\ (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{isHigh}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{isHigh}' = \mathsf{isHighTmp}) \\
\vee & (\mathsf{pc}_1 = 10 \wedge \mathsf{pc}_1' = 11) \wedge (\mathsf{pc}_0' = \mathsf{pc}_0) & \wedge\ (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{filter}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{filter}' = \mathsf{isXray}) \\
\vee & (\mathsf{pc}_1 = 11 \wedge \mathsf{pc}_1' = 12) \wedge (\mathsf{pc}_0' = \mathsf{pc}_0) & \wedge\ (\bigwedge_{\mathsf{var} \in X \setminus \{\mathsf{highEnergy}, \mathsf{pc}_0, \mathsf{pc}_1\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & (\mathsf{highEnergy}' = \mathsf{isHigh})
\end{array}
$$

Fig. 1: Transition relation for Listing 1.1

Example 5. The floating point program from Listing 1.2 can be modeled as an STS where $X = \{\mathsf{v}, \mathsf{y}, \mathsf{pc}, \mathsf{print}\}$, $I = \emptyset$, the initial condition is $(\mathsf{pc} = 3 \wedge \mathsf{v} = 10^{308} \wedge \mathsf{y} = 0)$, the final condition is $\mathsf{pc} = 5$, and the transition relation is defined as $(\mathsf{pc} = 3 \wedge \mathsf{pc}' = 5 \wedge \mathsf{y}' = \mathsf{v} * \mathsf{v} \wedge \mathsf{v}' = \mathsf{v} \wedge \neg\mathsf{print} \wedge \mathsf{print}' = \mathsf{print}) \vee (\mathsf{pc} = 3 \wedge \mathsf{pc}' = 5 \wedge \mathsf{y}' = \mathsf{convert64}(\mathsf{v} * \mathsf{v}) \wedge \mathsf{v}' = \mathsf{v} \wedge \mathsf{print} \wedge \mathsf{print}' = \mathsf{print})$ where $\mathsf{convert64}$ is a function producing the 64 bit representation of the number. The initial condition does not constrain print, i.e., the initial value of print can be arbitrary; this initial (nondeterministic) choice then fixes the respective disjunct of the transition relation depending on whether the `printf`-statement is present or not.

In the following, we formally define a number of useful properties of STSs.

Definition 2 (Termination). *An STS is* terminating *if the STS does not have infinite traces.*

Definition 3 (Input Determinism). *An STS is* input-deterministic *if 1) for every input i there is at most one state s such that $\langle s, i\rangle \models \mathsf{init}$, and 2) for every state s and every input i there is at most one successor state. Otherwise, it is* nondeterministic*.*

Definition 4 (Input-enabled). *An STS is* input-enabled *if 1) for every input i there is at least one state s such that $\langle s, i\rangle \models \mathsf{init}$, and 2) every configuration that is not final has at least one successor. In case 2) is violated, we call the transition relation* partial*.*

We next define assertions as well as succeeding and failing executions:

Definition 5 (Assertions, Succeeding and Failing Executions). *An* assertion *is a formula φ over the system variables X. An execution $\pi \stackrel{\text{def}}{=} c_0, \ldots, c_n$* succeeds *with respect to φ if $c_n|_X \models \varphi$. Similarly, π* fails *if $c_n|_X \models \neg\varphi$. Abusing our notation, we write $\pi \models \varphi$ if π succeeds and $\pi \not\models \varphi$ if π fails.*

We note that without input-enabledness, which we do not require in general, traces can get stuck at non-final configurations: For example, in Figure 1, any state with $\mathsf{pc}_0 = 5, \mathsf{pc}_1 = 12, \mathsf{isXray} = \mathsf{false}, \mathsf{input}_2 = \mathsf{true}$ does not have a

control flow		inputs		states					
pc_0	pc_1	input_1	input_2	isXray	isHigh	isHighTmp	filter	highEnergy	
4	10	F	τ	T	T	F	F	F	
5	10	τ	F	F	T	F	F	F	
7	10	τ	τ	F	T	F	F	F	
8	10	τ	τ	F	F	F	F	F	
8	11	τ	τ	F	F	F	F	F	
8	12	τ	τ	F	F	F	F	F	steps

Fig. 2: Execution π_1 of Figure 1

control flow		inputs		states					
pc_0	pc_1	input_1	input_2	isXray	isHigh	isHighTmp	filter	highEnergy	
4	10	F	τ	T	T	F	F	F	
5	10	τ	F	F	T	F	F	F	
7	10	τ	τ	F	T	F	F	F	
7	11	τ	τ	F	T	F	F	F	
7	12	τ	τ	F	T	F	F	T	
8	12	τ	τ	F	F	F	F	T	steps

Fig. 3: Execution π_2 of Figure 1

successor. For such traces, it is not meaningful to argue whether they satisfy an assertion. This is why Definition 5 quantifies over executions, i.e., traces that end in a final configuration. Moreover, Definition 5 disregards infinite traces, as we limit ourselves in this paper to Heisenbugs that are observable in a finite amount of time; we leave the extension to non-terminating traces to future work.

Example 6. Figures 2 and 3 show executions of the STS from Example 4. For $\varphi \stackrel{\text{def}}{=} (\mathsf{filter} \vee \neg \mathsf{highEnergy})$ (the assertion in line 13), we have $\pi_1 \models \varphi$ and $\pi_2 \not\models \varphi$.

We presume that a system contains a bug if it has at least one failing execution (i.e., we assume that the assertion φ correctly encodes desired behavior of the program):

Definition 6. *Let $(X, I, \mathsf{init}, \mathsf{final}, T)$ be an STS and the assertion φ be a formula over X. The STS contains a* bug *with respect to φ if there exists a failing counterexample execution:*

$$\exists \pi_c \,.\, \pi_c \not\models \varphi.$$

A violation of the property φ in Definition 6, however, does not necessarily constitute a Heisenbug.

2.2 Formal Definition of Heisenbugs

Heisenbugs are special bugs which occur only on some, but not on all executions. We express this in terms of a hyperproperty [7]. Unlike properties over single executions (such as Definition 6), hyperproperties relate sets of traces, allowing us to characterize Heisenbugs by juxtaposing the behavior of two executions. In particular, we require at least one succeeding and one failing execution induced by the same input (as deviating behavior is to be expected for differing inputs). To express this requirement for reactive systems, we define the projection of a trace to its corresponding sequence of inputs that are not quiescent (i.e., not τ):

Definition 7. *Let π be a trace of the STS $(X, I, \mathsf{init}, \mathsf{final}, T)$, and let $J \subseteq I$ be some subset of the input variables. The* input sequence *$J(\pi)$ is defined inductively:*

$$J(\varepsilon) = \varepsilon, \qquad J(c \cdot \pi) = \begin{cases} J(\pi), & \text{if } \forall \mathsf{i} \in J : c(i) = \tau \\ c|_J \cdot J(\pi) & \text{otherwise} \end{cases}$$

where ε is the trace of length zero and $\cdot$ represents concatenation.

Example 7. The traces from Figures 2 and 3 have the same inputs $\langle \mathsf{input}_1 \mapsto \mathsf{false}, \mathsf{input}_2 \mapsto \tau \rangle \cdot \langle \mathsf{input}_1 \mapsto \tau, \mathsf{input}_2 \mapsto \mathsf{false} \rangle$ for $J = I = \{\mathsf{input}_1, \mathsf{input}_2\}$.

Heisenbugs can be characterized using a hyperproperty asserting the existence of two executions with matching inputs, one of which violates the assertion while the other fulfills it:

Definition 8. *An STS $(X, I, \mathsf{init}, \mathsf{final}, T)$ contains a* Heisenbug *with respect to an assertion φ if*

$$\exists \pi_c, \pi_w \,.\, I(\pi_c) = I(\pi_w) \wedge \pi_c \not\models \varphi \wedge \pi_w \models \varphi.$$

The execution π_c is the counterexample execution*, π_w is the* witness execution*.*

We emphasize that the definition is expressed in terms of a hyperproperty stating that the inputs of the two traces must match. This condition cannot be expressed as a simple trace property. Moreover, we remark that Definition 8 is amenable to hyperproperty model checking (e.g., [13]).

Example 8. The Therac-25 example contains a Heisenbug with counterexample execution π_2 from Figure 3 and witness execution π_1 from Figure 2.

3 Causality

In this section, we extend the hyperproperty from Definition 8 to counterfactually reason about the causality of Heisenbugs. We first present a refinement step for making potential causes explicit in the model and then introduce formal definitions of causality in deterministic as well as nondeterministic systems.

3.1 Modeling Sources of Nondeterminism

For the purpose of causality analysis, the sources of nondeterminism (which we call *mechanisms*) need to be made explicit. Nondeterminism can be due to incomplete observability, incomplete modeling or to inherent stochasticity in the modeled system, as is the case for example in quantum mechanics [15, Section 3.1]. Nondeterminism stemming from incomplete observability and modeling can be eliminated by refining the model with the relevant information. Even true nondeterminism can—at least in principle—be accounted for by means of prophecy variables [1].

To formalize this idea, we introduce refinements of a transition system:

$$
\begin{array}{llll}
 & \overbrace{\neg\mathsf{thread}}^{\text{schedule}} & \overbrace{\wedge(\mathsf{pc}_0 = 4 \wedge \mathsf{pc}'_0 = 5) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1)}^{\text{control flow}} & \wedge \quad \overbrace{\ldots}^{\text{data flow}} \\
\vee & \neg\mathsf{thread} & \wedge(\mathsf{pc}_0 = 5 \wedge \mathsf{pc}'_0 = 7) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge \quad \ldots \\
\vee & \neg\mathsf{thread} & \wedge(\mathsf{pc}_0 = 7 \wedge \mathsf{pc}'_0 = 8) \wedge (\mathsf{pc}'_1 = \mathsf{pc}_1) & \wedge \quad \ldots \\
\vee & \mathsf{thread} & \wedge(\mathsf{pc}_1 = 10 \wedge \mathsf{pc}'_1 = 11) \wedge (\mathsf{pc}'_0 = \mathsf{pc}_0) & \wedge \quad \ldots \\
\vee & \mathsf{thread} & \wedge(\mathsf{pc}_1 = 11 \wedge \mathsf{pc}'_1 = 12) \wedge (\mathsf{pc}'_0 = \mathsf{pc}_0) & \wedge \quad \ldots
\end{array}
$$

Fig. 4: Deterministic transition relation for Listing 1.1

Definition 9 (Refinement). *Let $S \stackrel{\text{def}}{=} (X, I, \mathsf{init}, \mathsf{final}, T)$ be an STS. We say an STS $S_{\mathsf{ref}} = (X \uplus X_{\mathsf{ref}}, I \uplus I_{\mathsf{ref}}, \mathsf{init}_{\mathsf{ref}}, \mathsf{final}_{\mathsf{ref}}, T_{\mathsf{ref}})$ is a* refinement *of S iff*

1. *for every $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle, \langle s', s'_{\mathsf{ref}}\rangle\rangle \models T_{\mathsf{ref}}$ we have that $\langle s, i, s'\rangle \models T$, and for every state $\langle s, s_{\mathsf{ref}}\rangle$ of S_{ref} and transition $\langle s, i, s'\rangle \models T$ there are mappings $i_{\mathsf{ref}}, s'_{\mathsf{ref}}$ of I_{ref} and X'_{ref} to values such that $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle, \langle s', s'_{\mathsf{ref}}\rangle\rangle \models T_{\mathsf{ref}}$,*
2. *for every $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle\rangle \models \mathsf{init}_{\mathsf{ref}}$ we have $\langle s, i\rangle \models \mathsf{init}$, and for every $\langle s, i\rangle \models \mathsf{init}$ there are mappings $i_{\mathsf{ref}}, s_{\mathsf{ref}}$ of I_{ref} and X_{ref} to values such that $\langle\langle s, s_{\mathsf{ref}}\rangle, \langle i, i_{\mathsf{ref}}\rangle\rangle \models \mathsf{init}_{\mathsf{ref}}$, and*
3. *for every $\langle s, s_{\mathsf{ref}}\rangle \models \mathsf{final}_{\mathsf{ref}}$ we have $s \models \mathsf{final}$, and for every $s \models \mathsf{final}$ and every mapping s_{ref} of the variables X_{ref} to values we have $\langle s, s_{\mathsf{ref}}\rangle \models \mathsf{final}_{\mathsf{ref}}$.*

We note that the above definition preserves executions: Let S_{ref} be a refinement of some STS S. Then every execution of S_{ref} gives rise to an execution of S by projecting away the additional state and input variables. On the other hand, the conditions in the refinement definition ensure that every execution of S can be extended to an execution of S_{ref} by choosing suitable values for the additional state and input variables. We note that refinements can be thought of as adding additional information to the STS under analysis, and the requirements in our definition ensure that executions are preserved. If *all* mechanisms (i.e., sources of nondeterminism) are explicit, refinement yields a deterministic system:

Definition 10 (Determinization). *We say that an STS S_{det} is a* determinization *of some STS S, if S_{det} is a refinement of S and is input-deterministic.*

Example 9. The Therac-25 transition system from Example 4 can be refined by setting $I_{\mathsf{ref}} = \{\mathsf{thread}\}$, where thread is a Boolean variable selecting which thread takes a step. The refined transition relation is shown in Figure 4.

Example 10. The floating point transition system from Example 5 can be extended to a deterministic system by setting $I_{\mathsf{ref}} = \{\mathsf{debug}\}$ and considering the refined initial condition $(\mathsf{pc} = 3 \wedge \mathsf{v} = 10^{308} \wedge \mathsf{y} = 0 \wedge (\mathsf{debug} \Leftrightarrow \mathsf{print}))$, and leaving the transition relation unchanged. We point out that the initial value of the input variable debug fixes the value of print, which in turn fixes the transition relation reflecting the presence of the `printf`-statement.

For Example 9 and Example 10 we have $X_{\mathsf{ref}} = \emptyset$. In the Peterson example below, the refinement contains a state variable reflecting whether the cache state has been propagated to main memory.

$$
\begin{array}{llll}
 & \overbrace{\text{control flow}} & & \overbrace{\text{data flow}} \\
 & ((\mathsf{pc}_0 = 5 \wedge \mathsf{pc}_0' = 6) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{flagP0c}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{flagP0c}' = 1) \wedge \\
 & & & (\mathsf{delay} \wedge \mathsf{flagP0}' = \mathsf{flagP0} \vee \neg\mathsf{delay} \wedge \mathsf{flagP0}' = 1) \\
\vee & (\mathsf{pc}_0 = 6 \wedge \mathsf{pc}_0' = 7) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{turn}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{turn}' = 1) \\
\vee & (\mathsf{pc}_0 = 7 \wedge \mathsf{pc}_0' = 8) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{print} \Rightarrow \neg\mathsf{delay}) \\
\vee & (\mathsf{pc}_0 = 8 \wedge \mathsf{pc}_0' = 9) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V} \mathsf{var}' = \mathsf{var} \wedge (\neg\mathsf{flagP1} \vee \mathsf{turn} = 0) \\
\vee & (\mathsf{pc}_0 = 9 \wedge \mathsf{pc}_0' = 10) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{critical}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & & (\mathsf{critical}' = \mathsf{critical} + 1) \wedge \\
 & & & (\neg\mathsf{delay} \wedge \mathsf{flagP0}' = \mathsf{flagP0} \vee \mathsf{delay} \wedge \mathsf{flagP0}' = \mathsf{flagP0c}) \\
\vee & (\mathsf{pc}_0 = 10 \wedge \mathsf{pc}_0' = 11) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{error}\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & & (\mathsf{critical} \neq 1 \Rightarrow \mathsf{error}' = \mathsf{error} + 1) \wedge \\
 & & & (\mathsf{critical} = 1 \Rightarrow \mathsf{error}' = \mathsf{error}) \\
\vee & (\mathsf{pc}_0 = 11 \wedge \mathsf{pc}_0' = 12) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{critical}\}} \mathsf{var}' = \mathsf{var}) \wedge (\mathsf{critical}' = \mathsf{critical} - 1) \\
\vee & (\mathsf{pc}_0 = 12 \wedge \mathsf{pc}_0' = 13) \wedge (\mathsf{pc}_1' = \mathsf{pc}_1) & \wedge & (\bigwedge_{\mathsf{var} \in V \setminus \{\mathsf{flagP0c}, \mathsf{flagP0}\}} \mathsf{var}' = \mathsf{var}) \wedge \\
 & & & (\mathsf{flagP0c}' = 0) \wedge (\mathsf{flagP0}' = 0))
\end{array}
$$

Fig. 5: A part of T_{ref} for Listing 1.3 (where $V \stackrel{\text{def}}{=} (X \cup X_{\mathsf{ref}}) \setminus \{\mathsf{pc}_0, \mathsf{pc}_1\}$)

Example 11. Peterson's algorithm (Listing 1.3) can be modeled as a determinisitic STS. In this example we present the final refinement that makes all involved mechnisms explicit. Alternatively, the mechanisms could be made explicit in successive refinement steps. Figure 5 shows the part of the transition relation that models P0. Let $X = \{\mathsf{pc}_0, \mathsf{pc}_1, \mathsf{flagP0}, \mathsf{flagP0c}, \mathsf{flagP1}, \mathsf{flagP1c}, \mathsf{turn}, \mathsf{critical}, \mathsf{error}, \mathsf{print}\}$ where flagP0c and flagP1c represent the locally cached versions of the flags. We have $I = \emptyset$ and $I_{\mathsf{ref}} = \{\mathsf{thread}, \mathsf{debug}, \mathsf{reorder}\}$, where thread indicates whether P0 or P1 takes a step (thread is omitted in Figure 5). Let $X_{\mathsf{ref}} = \{\mathsf{delay}\}$, and let $\mathsf{init}_{\mathsf{ref}}$ imply $(\mathsf{print} = \mathsf{debug} \wedge \mathsf{delay} = \mathsf{reorder})$. The variable print indicates that the program version with `printf`-debugging is executed, and delay is true if the modifications of the flags flagP0 and flagP1 are only committed to shared memory after entering the critical section (to avoid clutter, we assume only two possible points for committing the modification of flagP0). We use $(\mathsf{print} \Rightarrow \neg\mathsf{delay})$ to model the interplay between two mechanisms where the `printf` instruction prevents reordering because of the added barrier, resulting in a partial transition relation. Moreover, $\mathsf{init}_{\mathsf{ref}}$ ensures that $\mathsf{flagP0} = \mathsf{flagP0c} = \mathsf{flagP1} = \mathsf{flagP1c} = \mathsf{turn} = \mathsf{critical} = \mathsf{error} = 0$ and $\mathsf{pc}_0 = 5$ and $\mathsf{pc}_1 = 15$, and $\mathsf{final}_{\mathsf{ref}}$ is $\mathsf{pc}_0 = 13 \wedge \mathsf{pc}_1 = 23$.

Note that the processor running the original nondeterministic version of Peterson's algorithm already has micro-architectural features that facilitate instruction reordering (not modeled in Example 11); the auxiliary input reorder and variable delay merely make this mechanism observable.

3.2 Defining Causes

In the following, we provide a formal definition of causes inspired by Lewis' counterfactuals [26] and the causality framework of Galles and Pearl [14].

Definition 11 (Cause). *Let* $S \stackrel{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ *be a deterministic STS, where* I *and* M *are disjoint sets of inputs, and let* φ *be an assertion. Let* $M = M_C \uplus M_N$*. We say that* M_C *is a* cause *with respect to* M *and* φ *and iff*

$$\exists \pi_c, \pi_w \,.\, I(\pi_c) = I(\pi_w) \wedge M_N(\pi_c) = M_N(\pi_w) \wedge \pi_c \not\models \varphi \wedge \pi_w \models \varphi \qquad (1)$$

and M_C *is a minimal subset of* M *with this property.*

We note that in the above definition we require the inputs I to agree on the executions π_c and π_w, while only the inputs M may differ. The rationale is that we want to apply this definition for studying the causes of Heisenbugs: We are given some (nondeterministic) STS with inputs I, which has a Heisenbug. We now consider some determinization of the STS to which we have added inputs M, modelling the *mechanisms* responsible for the nondeterminism. The above definition then allows to study the cause among the modelled mechanisms: A subset $M_C \subseteq M$ is a cause of a Heisenbug, if the Heisenbug still occurs when the inputs M_N agree in the deviating executions π_c and π_w.

Proposition 1 (Existence of a Cause). *Let* $S \stackrel{\text{def}}{=} (X, I, \mathsf{init}, \mathsf{final}, T)$ *be a nondeterministic STS with a Heisenbug (Definition 8) with respect to an assertion* φ *and let* $S_{\mathsf{det}} \stackrel{\text{def}}{=} (X \cup X_{\mathsf{det}}, I \cup M, \mathsf{init}_{\mathsf{det}}, \mathsf{final}_{\mathsf{det}}, T_{\mathsf{det}})$ *be a determinization of* S*. Then there exists a cause* M_C *with respect to* M *and* φ*.*

Proof. Let π_c and π_w be executions of S that satisfy Definition 8. Since refinements preserve executions, there must be executions $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ of S_{det} such that $\pi_{c\mathsf{det}}|_{(I \cup X)} = \pi_c$ and $\pi_{w\mathsf{det}}|_{(I \cup X)} = \pi_w$. Now assume that $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ agree on M (in addition to I). Let $\langle s_c, s_{c\mathsf{det}} \rangle$ and $\langle s_w, s_{w\mathsf{det}} \rangle$ be the initial states of $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$, respectively. Since S_{det} is input-deterministic, however, there is at most one state $\langle \langle s, s_{\mathsf{det}} \rangle, \langle i, m \rangle \rangle \models \mathsf{init}_{\mathsf{det}}$, hence $\langle s_c, s_{c\mathsf{det}} \rangle = \langle s_w, s_{w\mathsf{det}} \rangle$. Moreover, for every state $\langle s, s_{\mathsf{det}} \rangle$, each input $\langle i, m \rangle$ determines a unique successor state $\langle s', s'_{\mathsf{det}} \rangle$. Since $\pi_{c\mathsf{det}}|_{(I \cup M)} = \pi_{w\mathsf{det}}|_{(I \cup M)}$, this violates the assumption that $\pi_{c\mathsf{det}} \not\models \varphi$ and $\pi_{w\mathsf{det}} \models \varphi$. Hence, $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ must deviate on M. □

Example 12. The Peterson example contains a Heisenbug with respect to $\varphi \stackrel{\text{def}}{=} (\mathsf{error} = 0)$. Here, $\{\mathsf{reorder}\}$ and $\{\mathsf{thread}\}$ are causes, but $\{\mathsf{debug}\}$ is not: The set $\{\mathsf{reorder}\}$ is a cause because of two executions which both have $\mathsf{debug} = \mathsf{false}$ and the same schedule interleaving the critical sections, but only one execution sets $\mathsf{reorder} = \mathsf{true}$ and hence exhibits the bug. The set $\{\mathsf{thread}\}$ is a cause because of two executions which both have $\mathsf{debug} = \mathsf{false}$ and $\mathsf{reorder} = \mathsf{true}$ where one execution uses a sequential schedule of the two processes and the second execution uses a schedule interleaving the critical sections. Only the second execution exhibits the bug. However, the set $\{\mathsf{debug}\}$ is not a cause because any two executions would either both have to set $\mathsf{reorder} = \mathsf{false}$, making the bug impossible or both set $\mathsf{reorder} = \mathsf{true}$. In this case, by counterposition the constraint $(\mathsf{print} \Rightarrow \neg\mathsf{delay})$ enforces $\mathsf{debug} = \mathsf{false}$, yielding a bug on both executions if the schedule interleaves the critical sections or on no execution otherwise.

3.3 Causes and Nondeterminism

By introducing the notion of a *contributing cause* below, we show that even in the presence of nondeterminism we can still provide guarantees.

Definition 12 (Contributing Cause). *Let $S \stackrel{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ be a (potentially nondeterministic) STS, where I and M are disjoints set of inputs, and let φ, $M = M_C \uplus M_N$ satisfy the conditions in Definition 11. We call M_C a* contributing cause *of a Heisenbug.*

We argue that any contributing cause must be a subset of a cause in a corresponding determinization:

Theorem 1. *Let $S \stackrel{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ be a nondeterministic STS and let $S_{\mathsf{det}} \stackrel{\text{def}}{=} (X \cup X_{\mathsf{det}}, I \cup M \cup J, \mathsf{init}_{\mathsf{det}}, \mathsf{final}_{\mathsf{det}}, T_{\mathsf{det}})$ be a determinization of S. Let M_C be a contributing cause in S with respect to M and assertion φ. Then, there exists a cause C in S_{det} with respect to $M \cup J$ and φ such that $M_C \subseteq C \setminus J$.*

Proof. Consider two executions π_c and π_w satisfying Definition 12 for S. Since refinement preserves executions, there must be executions $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ in S_{det} such that $\pi_{c\mathsf{det}}|_{(I\cup M\cup X)} = \pi_c$ and $\pi_{w\mathsf{det}}|_{(I\cup M\cup X)} = \pi_w$ and $\pi_{c\mathsf{det}} \not\models \varphi$ and $\pi_{w\mathsf{det}} \models \varphi$. By Definition 12, for $M_N = M \setminus M_C$ it holds that $\pi_c|_{(I\cup M_N)} = \pi_w|_{(I\cup M_N)}$ and hence also $\pi_{c\mathsf{det}}|_{(I\cup M_N)} = \pi_{w\mathsf{det}}|_{(I\cup M_N)}$. Hence (following an argument similar to the one for Proposition 1) we argue that $\pi_{c\mathsf{det}}$ and $\pi_{w\mathsf{det}}$ must deviate on a subset of $M_C \cup J$, i.e., there exists a cause C satisfying Definition 11 such that $C \subseteq M_C \cup J$. Now assume that $M_C \not\subseteq C$. Then M_C is not minimal, since $(M_C \cap C)$ also constitutes a contributing cause. Thus, we must have $M_C \subseteq C \setminus J$. □

Example 13. The refined STS in Example 9 is nondeterministic as the initial values of filter and $\mathsf{highEnergy}$ are unconstrained. Following Definition 12, $\{\mathsf{thread}\}$ is a contributing cause. Consider a further refinement with $I_{\mathsf{ref}} = \{\mathsf{initF}, \mathsf{initH}\}$ and $\mathsf{init} = (\mathsf{filter} = \mathsf{initF} \wedge \mathsf{highEnergy} = \mathsf{initH} \wedge \mathsf{isXray} \wedge \mathsf{isHigh} \wedge \mathsf{pc}_0 = 4 \wedge \mathsf{pc}_1 = 10)$. As the initial values are never read, the cause is again $\{\mathsf{thread}\}$.

We provide a condition under which contributing causes are also causes:

Definition 13 (Cause in Presence of Nondeterminism). *Consider a (potentially nondeterministic) STS $S \stackrel{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$ such that for all traces π, π' of S with $\pi|_{I\cup M} = \pi'|_{I\cup M}$ we have that*

1. *π ends in a final state if and only if π' ends in a final state,*
2. *$\pi \models \varphi$ if and only if $\pi' \models \varphi$ (in case both traces end in a final state).*

Let φ, $M = M_C \uplus M_N$ satisfy the conditions in Definition 11. We say that M_C is a cause in presence of nondeterminism *with respect to M and φ.*

We will next state a justification for the introduction of the above definition. We first establish that input-enabled determinizations always exist:

Proposition 2. *Let* $S \stackrel{\text{def}}{=} (X, I, \mathsf{init}, \mathsf{final}, T)$ *be an input-enabled STS. Then, a deterministic input-enabled refinement* S_{ref} *always exists.*

Proof. We set $I_{\mathsf{ref}} = \{\mathsf{oracle}\}$ for a single variable oracle, whose values are mappings of configurations to successors, i.e., oracle fixes a successor state s' for every configuration $\langle s, i\rangle$ such that $\langle s, i, s'\rangle \models T$ (note that at least one successor state s' always exists because of our assumption that S is input-enabled). We then adopt T_{ref} from T as the transition relation that moves to the successor state fixed by the oracle variable. Likewise, we adopt the initial condition $\mathsf{init}_{\mathsf{ref}}$. □

We next establish that no matter the input-enabled determinization S' of an STS S, a cause in the presence of nondeterminism in S is always a cause in S'. Together with Proposition 2, which guarantees the existence of an input-enabled determinization at least in theory, we obtain that a cause in presence of nondeterminism can indeed by considered as a cause.

Theorem 2. *Let* M_C *be a cause in presence of nondeterminism with respect to mechanisms* M *in an STS* $S \stackrel{\text{def}}{=} (X, I \cup M, \mathsf{init}, \mathsf{final}, T)$*. Let* $S_{\mathsf{det}} \stackrel{\text{def}}{=} (X \cup X_{\mathsf{det}}, I \cup M \cup J, \mathsf{init}_{\mathsf{det}}, \mathsf{final}_{\mathsf{det}}, T_{\mathsf{det}})$ *be an input-enabled determinization of* S*. Then* M_C *is also a cause in* S_{det} *with respect to* $(I \cup J)$*.*

Proof. Let π_c and π_w be executions of S that satisfy Definition 13. Since refinements preserve executions, there must be an execution $\pi_{c\mathsf{det}}$ of S such that $\pi_{c\mathsf{det}}|_{(I \cup M \cup X)} = \pi_c$. In particular, we have $\pi_{c\mathsf{det}} \not\models \varphi$. Because S_{det} is input-enabled we can obtain a trace π of S_{det} such that $\pi|_J = \pi_{c\mathsf{det}}|_J$ and $\pi|_{I \cup M} = \pi_w|_{I \cup M}$. Note that π induces a trace π' of S with $\pi' = \pi_{w\mathsf{det}}|_{I \cup M \cup X}$. Hence, by the assumptions stated in Definition 13, the trace π' is in fact an execution (i.e., ends with a final configuration), and we have $\pi' \models \varphi$. Thus, we also get that π is an execution and that we have $\pi \models \varphi$. □

Example 14. The nondeterministic refinement of the Therac-25 STS in Example 9 satisfies the properties in Definition 13. The refinement in Example 13 is input-enabled and deterministic and the contributing cause is indeed a cause.

3.4 Testing and Causal Analysis

In the context of testing, an evaluation of Definition 11 and Definition 12, respectively, is limited to the subset of the executions induced by a given test suite. Lemma 1 characterizes the results that can be drawn by analyzing a subset of the executions of an STS:

Lemma 1. *Let* π_c *and* π_w *be executions satisfying Equation 1 in Definition 11 (or Definition 12, respectively) and let* M_C *be the inputs deviating in* π_c *and* π_w*. Then* M_C *is a superset of a cause (or contributing cause, respectively).*

Proof. Note that M_C is a cause according to Definition 11 (or a contributing cause according to Definition 12) if it is minimal with respect to Equation 1. Otherwise, there must be a cause that is a subset of M_C. □

Lemma 1 provides guarantees even if an exhaustive analysis is infeasible. If, in addition, the conditions in Definition 13 are met (i.e., we can control or at least observe the relevant mechanisms), then Proposition 2, Theorem 2, and Lemma 1 guarantee that each overapproximation of a cause identified by testing includes a *non-empty* (contributing) cause.

4 Analysis Methodology and Challenges

We sketch an (iterative) methodology for practical analyses based on the formalization above and showcase two possible instantiations and their challenges:

① **Task:** Starting from a Heisenbug (Definition 8), identify candidate mechanisms M (e.g., consulting surveys [31]).
Challenge: The accuracy of the analysis is contingent on identifying the relevant mechanisms.

② **Task:** Pick a mechanism $\mathsf{m} \in M$ and adapt (or refine according to Definition 9) the model or system to make m controllable (or at least observable).
Challenge: The system may be inherently uncontrollable or unobservable, or attempts to control/observe it potentially introduce a probe effect.

③ **Task:** Identify (contributing) causes by finding witnesses that deviate in as few mechanisms as possible (i.e., satisfy Equation 1 in Definition 11).
Challenge: Testing will yield over-approximations only (cf. Lemma 1).

④ **Task:** Check a stopping criterion to determine whether further mechanisms or refinement steps are required (steps ① and ②).
Challenge: Assessing whether all causes have been correctly identified is challenging and may amount to fixing the bug and re-verifying the system.

Causal Analysis based on Model Checking. We built a NuSMV [5] model of Peterson's algorithm (Listing 1.3). We use self-composition [3], which composes two copies S_w and S_c of the STS S, to reduce the existence of a counterexample trace and a witness trace (which is a hyperproperty) to the existence of a single trace in the composed model. NuSMV can then construct the trace as a counterexample to an LTL property over the composed model. As NuSMV usually considers infinite traces, final conditions are accounted for in the property. The existence of a Heisenbug can be confirmed by checking that NuSMV finds a counterexample to the property $\psi := G(\mathsf{final}_c \wedge \mathsf{final}_w \Rightarrow (\varphi_w \Rightarrow \varphi_c))$ for final and φ as in Example 11 and Example 12 (where subscripted predicates range over the matching variable set).

In step ①, we pick the fact whether the print statements are executed and model it adding variables print_w and print_c to the model (step ②). In step ③, we invoke NuSMV on the property $G(\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \Rightarrow \psi$. As there is a counterexample, we identify the empty set as a contributing cause.

We start another refinement iteration, pick concurrency as machanism (step ①) and model it by variables thread_w and thread_c (step ②). We check the property $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c)) \Rightarrow \psi$ (step ③). Again, there is a counterexample and the empty set is a contributing cause.

```
1  bool flag0 = false;
2  bool flag1 = false;
3  spinlock_t lock0, lock1;
4  void *thread0(void*) {
5    spin_lock(lock0);
6      flag0 = true;
7      assert (!flag1);
8      yield();
9      spin_lock(lock1);
10       flag0 = false;
11     spin_unlock(lock1);
12     yield();
13   spin_unlock(lock0);
14 }
```

```
15 void *thread1(void*) {
16   spin_lock(lock1);
17     flag1 = true;
18     assert (!flag0);
19     yield();
20     spin_lock(lock0);
21       flag1 = false;
22     spin_unlock(lock0);
23     yield();
24   spin_unlock(lock1);
25 }
```

Listing 1.4: An assertion fails if (and only if) a deadlock occurs.

In the next refinement iteration, we pick the weak memory behavior (step ①) we model it by variables delay_w and delay_c and reflect the fact that $\mathsf{print} \implies \neg\mathsf{delay}$ (cf. Example 11) (step ②). Checking property $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c) \wedge (\mathsf{delay}_w \Leftrightarrow \mathsf{delay}_c)) \Rightarrow \psi$ returns true, hence we have now found a non-empty cause superset and can start cause minimization. A counterexample to $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c)) \Rightarrow \psi$ witnesses that delay is a cause, similarly a counterexample to $G((\mathsf{print}_w \Leftrightarrow \mathsf{print}_c) \wedge (\mathsf{delay}_w \Leftrightarrow \mathsf{delay}_c)) \Rightarrow \psi$ witnesses that thread is a cause. As the model satisfies $G((\mathsf{delay}_w \Leftrightarrow \mathsf{delay}_c) \wedge (\mathsf{thread}_w \Leftrightarrow \mathsf{thread}_c)) \Rightarrow \psi$, print is not a cause. This concludes step ③. As we identified a non-empty cause, no more refinement steps are needed.

Test-based Causal Analysis. Consider the code in Listing 1.4, which might deadlock because of a faulty locking discipline. The assertions in lines 7 and 18 fail when a deadlock, caused by a specific (combination of) context switche(s), occurs: a context switch at line 8 to `thread1` (or, symmetrically, from line 19 to `thread0`) causes both threads to wait for a lock held by the other thread.

In step ①, we identify concurrency (limited to the context switches marked by `yield` for simplicity) as potential cause. Following the approach of KISS [32], we control the scheduler (step ②) by sequentializing the concurrent program and simulating the execution of a large subset of its interleavings. In KISS, threads can be started and terminated nondeterministically at any point during the execution. Using closures to save the local state of a thread, we add the capability to *re-enter* a thread after its interruption by `yield`. The execution of `thread0` (`thread1`, respectively) can be interrupted at lines 8 and 12 (19 and 23, respectively). Our sequentialization enables us to explicitly control these four context switches, inducing 2^4 potential schedules. Random (or systematic) exploration of these schedules then yields executions that terminate normally or violate an assertion. Failing executions deviate from the non-failing ones by performing a context switch at lines 8 or 19, at least one of which must constitute (part of) the candidate cause(s) we identify in step ③.

Testing merely provides an over-approximation of the cause M_C (Lemma 1). Due to the minimality requirement in Definition 11 and Definition 12, however, removing one element from M_C (by controlling the mechanism accordingly) eliminates the entire cause. Assume for now, that `thread0` in Listing 1.4 always executes first, in which case the context switch at line 8 is a unique cause for the deadlock. Consider an over-approximation comprising of two context switches at lines 8 and 19. Blocking the context switch at line 8 eliminates the Heisenbug, while blocking the one at line 19 doesn't. By individually blocking the context switches and checking whether subsequent testing provides sufficient confidence that the bug has been eliminated, we obtain a stopping criterion in step ④.

If, however, executions may start with `thread0` or `thread1`, the context switches at lines 8 and 19 form two independent (non-intersecting) causes (due to the symmetry in Listing 1.4). Consequently, *both* context switches must be identified to eliminate all causes of the bug (cf. Section 3.4). Blocking individual context switches (as suggested above) does not provide a reliable stopping criterion. Despite this limitation, testing-based analysis can help the developer to narrow down the set of candidate causes significantly.

5 Related Work

Terminology and Definition of Heisenbugs. The first paper mentioning Heisenbugs [17] uses the term for transient software bugs which disappear under observation. In [18], bugs are classified into Bohrbugs (bugs manifesting consistently), Mandelbugs (bugs with complex error propagation), and Heisenbugs (bugs manifesting differently under the probe effect). In contrast to this informal classification, our definition is formal, covering Heisenbugs which stem from the probe effect as well as from nondeterminism. The term is frequently (and informally) used in the context of concurrency [30], where it exclusively refers to bugs caused by control-flow nondeterminism. In the context of testing, the notion of flaky tests [31] resembles the notion of Heisenbugs. The comparison of failing and non-failing executions is used in several lines of research with goals orthogonal to the definition of bug classes. Differential assertion checking [21] compares failing and non-failing executions to define relative correctness of different program versions. In the context of diagnosability, the notion of critical pairs of failing and non-failing executions with equivalent observations is used to check whether faults can be detected at runtime [6].

Causality. Our definition of causality is inspired by Lewis' counterfactuals [25]. The negation of Definition 11 mirrors the definition of causal irrelevance in [14] and Definition 11 corresponds to its dual notion of causality between variables [12]. A core difference is that our interventions are restricted to inputs that represent nondeterministic mechanisms rather than affecting arbitrary points of the transition relation (or the causal model). Moreover, causal models have a fixed propagation depth, while we consider an arbitrary number of unwindings of the transition relation. Halpern and Pearl [20,19] provide a widely accepted definition of "actual" causes based on counterfactuals, where contingencies are used to

control interference between interventions. Several lines of work reason about the origin of system faults [23,2,10,4,16] using Halpern and Pearl's notion of causality. In [8], actual causality is used to explain violations of hyperproperties. It formalizes causes for violations of (arbitrary) universally quantified hyperproperties as a hyperproperty with quantifier alternation, which can then be checked with a model checker such as [13]. We formalize causes for Heisenbugs (a specific hyperproperty) in terms of an existentially quantified hyperproperty.

Several approaches exist for automatically detecting causes of flaky tests. The RootFinder tool [22] collects passing and failing executions and correlates their differences with a specific cause. In [39] the authors present a tool for finding code locations that lead to differences between succeeding and failing executions. Identifying what happens in these locations is left to the developer. In [36] and [29] the system is repeatedly executed under different configurations to check which configuration influences the manifestation of the bug. All of these approaches are based on computing correlations rather than performing rigorous causal inference. In contrast, our framework is based on a formal causal analysis accounting for interactions of multiple potential causes. [31] provides a taxonomy of causes relevant in the context of automated testing.

6 Conclusion

While the term Heisenbug is widely used, its exact meaning often depends on the context. We provide a formal definition that unifies the notion of Heisenbugs caused by a system alteration and those caused by nondeterminism. Furthermore, we present a hyperproperty-based framework for determining which mechanisms cause the manifestation of a Heisenbug. In particular, our approach allows the identification of causes in the presence of multiple mechanisms that could trigger a Heisenbug and gives guarantees for results of a causal analysis even in presence of nondeterminism. Building on this result, we sketch a methodology for causal analysis based on iterative refinement.

Acknowledgements This work was partially supported by ERC CoG ARTIST 101002685, by the FWF project W1255-N23, by a netidee scholarship, and by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG11005].

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS (1988)
2. Baier, C., Dubslaff, C., Funke, F., Jantsch, S., Majumdar, R., Piribauer, J., Ziemek, R.: From verification to causality-based explications. In: ICALP (2021)
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM (2011)
4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: CAV (2009)
5. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV (2002)
6. Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: IJCAI (2003)
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6) (2010)
8. Coenen, N., Dachselt, R., Finkbeiner, B., Frenkel, H., Hahn, C., Horak, T., Metzger, N., Siber, J.: Explaining hyperproperty violations. In: CAV (2022)
9. Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R., Trivedi, K.S.: Fault triggers in open-source software: An experience report. In: ISSRE (2013)
10. Dubslaff, C., Weis, K., Baier, C., Apel, S.: Causality in configurable software systems. In: ICSE (2022)
11. Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A.: Understanding flaky tests: The developer's perspective. In: ESEC/FSE (2019)
12. Eiter, T., Lukasiewicz, T.: Complexity results for structure-based causality. Artif. Intell. **142**(1) (2002)
13. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: CAV (2015)
14. Galles, D., Pearl, J.: Axioms of causal relevance. Artif. Intell. **97**(1-2) (1997)
15. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive computation and machine learning, MIT Press (2016)
16. Gössler, G., Stefani, J.B.: Causality analysis and fault ascription in component-based systems. Theor. Comput. Sci. **837** (2020)
17. Gray, J.: Why do computers stop and what can be done about it? Tech. Rep. 85.7, PN87614, Tandem Computers (June 1986)
18. Grottke, M., Trivedi, K.S.: A classification of software faults. In: ISSRE (2005)
19. Halpern, J.Y.: A modification of the halpern-pearl definition of causality. In: IJCAI (2015)
20. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural-model approach: Part 1: Causes. British Journal for the Philosophy of Science **56** (2005)
21. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: ESEC/FSE (2013)
22. Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S.: Root causing flaky tests in a large-scale industrial setting. In: ISSTA (2019)
23. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: VMCAI (2013)
24. Leveson, N., Turner, C.: An investigation of the Therac-25 accidents. IEEE Computer **26**(7) (1993)
25. Lewis, D.: Causation. Journal of Philosophy **70**(17) (1974)
26. Lewis, D.: Counterfactuals. Wiley-Blackwell (2001)

27. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access-interleaving invariants. IEEE Micro **27**(1) (2007)
28. Monniaux, D.: The pitfalls of verifying floating-point computations. TOPLAS **30**(3) (2008)
29. Moran, J., Augusto Alonso, C., Bertolino, A., de la Riva, C., Tuya, J.: Flakyloc: Flakiness localization for reliable test suites in web applications. JWE (2020)
30. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
31. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: A survey of flaky tests. TOSEM **31**(1) (2021)
32. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI (2004)
33. Ratliff, Z.B., Kuhn, D.R., Kacker, R.N., Lei, Y., Trivedi, K.S.: The relationship between software bug type and number of factors involved in failures. In: ISSRE Wksp (2016)
34. Senftleben, M.: Operational Characterization of Weak Memory Consistency Models. Master's thesis, University of Kaiserslautern (2013)
35. Sommerville, I.: Software Engineering. Addison-Wesley, 9 edn. (2010)
36. Terragni, V., Salza, P., Ferrucci, F.: A container-based infrastructure for fuzzy-driven root causing of flaky tests. In: ICSE (2020)
37. Thomas, M.: The story of the therac-25 in lotos. High Integrity Systems Journal (1994)
38. Tretmans, J.: Test generation with inputs, outputs, and quiescence. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (1996)
39. Ziftci, C., Cavalcanti, D.: De-Flake your tests : Automatically locating root causes of flaky tests in code at google. In: ICSME (2020)

Differential Property Monitoring for Backdoor Detection*

Otto Brechelmacher[1], Dejan Ničković[1], Tobias Nießen[2], Sarah Sallinger[2(✉)], and Georg Weissenbacher[2]

[1] Austrian Institute of Technology, Vienna, Austria firstname.lastname@ait.ac.at
[2] TU Wien, Vienna, Austria firstname.lastname@tuwien.ac.at

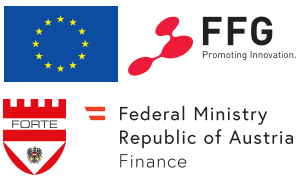
Abstract. A faithful characterization of backdoors is a prerequisite for an effective automated detection. Unfortunately, as we demonstrate, formalization attempts in terms of temporal safety properties prove far from trivial and may involve several revisions. Moreover, given the complexity of the task at hand, a hapless revision of a property may not only eliminate but also *introduce* inaccuracies in the specification. We introduce a method called *differential property monitoring* that addresses this challenge by monitoring discrepancies between two versions of a property, and illustrate that this technique can also be used to analyze observations of untrusted components. We demonstrate the utility of the approach using a range of case studies – including the recently discovered `xz` backdoor.

1 Introduction

Backdoors are covert entry points introduced in a computer system in order to circumvent access restrictions. The notion recently made a prominent appearence in mainstream news [22] in form of a backdoor in the Linux utility `xz` (CVE-2024-3094), where a pseudonymous agent went to great lengths to maliciously implant remote execution capabilities in the `liblzma` library. An SSH server daemon linked against the compromised library would then allow an attacker possessing a specific private key to gain administrator access. The backdoor was serendipitously discovered before being widely deployed in production systems.

Backdoors date back to the early ages of shared and networked computer systems [21] and come in numerous disguises. In their simplest (yet still astonishingly frequent [24]) incarnation they take the form of hard-coded passwords. On the other end of the spectrum, the complexity of backdoors recently culminated in a backdoor in Apple devices involving a sophisticated attack chain that exploits four zero-day vulnerabilities in software as well as hardware [14].

* Funded through the European Union through the ERC CoG ARTIST 10100268, its Horizon 2020 research and innovation programme under grant agreement No 101034440, a netidee scholarship, and via the Defense Research Programme FORTE of the Austrian Federal Ministry of Finance and managed by the Austrian Research Promotion Agency (FFG).



```
void do_authentication2(
    struct ssh *ssh) {
  Authctxt *authctxt = ssh->authctxt;
  while (!authctxt->success) {
    ...
    if (sshkey_verify(...))
      authenticated = 1;
    ...
  }
}
```

```
int main(int ac, char **av) {
  struct ssh *ssh;
  ...
  do_authentication2(ssh);
  ...
  do_authenticated(ssh);
  ...
}
```

Listing 1.1: `sshd` authentication flow

Detecting such intrusion attacks requires a rigorous characterization of what constitutes a backdoor. However, due to their variety, a simple formal definition is elusive. Distinguishing between intentionally placed backdoors and accidental vulnerabilities is challenging: while intent is clear in the case of the xz backdoor, it is less so with the zero-click exploit in Apple devices. Although attempts to formalize intent have been made (e.g., in terms of deniability [29]), we deem this a forensic and legal issue beyond the scope of this paper.

Property Template and Instantiation. Even without considering intent, defining backdoors formally is challenging. Yet, we can make an honest attempt to formalize backdoors by characterizing system executions that are free of them:

$$\forall \mathsf{user} \,.\, \forall \mathsf{resource} \,.\, \mathbf{G}(\mathtt{access}(\mathsf{user}, \mathsf{resource}) \Rightarrow \mathtt{permission}(\mathsf{user}, \mathsf{resource})) \quad (1)$$

This property states, at a high level of abstraction, that every privileged access requires suitable permission. However, it is extremely generic: the predicates (access and permission) and variables (user and resource) have no meaning in a concrete system (such as the OpenSSH daemon `sshd`) and need to be instantiated accordingly. Instantiating the template in Equation 1 requires significant technical insight and discretion regarding which system components and observations can be trusted. As an example, Listing 1.1 shows the (simplified) authentication flow of `sshd`. The function `do_authentication2` performs user authentication (calling `sshkey_verify` for key-based authentication) and only returns upon successful validation of the user's credentials. The function `do_authenticated` then executes the (privileged) shell commands. Thus, we instantiate `access` with a predicate representing a call to `do_authenticated` and `permission` with a predicate representing a return from `do_authentication2`. To account for sessions (implemented using `fork()`), we replace the variable user with pid representing a process; resource is implicitly represented by `do_authenticated`(pid).

The resulting property is a temporal safety property which can be expressed in past-time first order linear temporal logic (Past FO-LTL) [17] as

$$\forall\ \mathsf{pid} \,.\, \mathbf{G}(\mathtt{do\_authenticated}(\mathsf{pid}) \Rightarrow \mathbf{O}\, \mathtt{do\_authentication2}(\mathsf{pid}))\,, \quad (2)$$

where **O** is a temporal operator expressing that something happened in the past.

Runtime Verification. The property in Equation 2 can then be checked using an appropriate analysis technique. We argue that runtime monitoring is best suited for this task. The `xz` backdoor mechanism was concealed in a binary deployed during the build process rather than in the library's source code, making static

code analyses ineffective. Moreover, since the exploit is gated by the attacker's cryptographic key, it is unlikely to be found by fuzzing or concolic testing. Finally, Past FO-LTL is supported by the DejaVu monitoring tool [17].

Property Refinement. At this point, we could conclude our exposition if not for one grave flaw of our property in Equation 2: it fails to detect the `xz` backdoor. This is because the `xz` backdoor is technically not an authentication bypass (which is a common definition of backdoors) but a remote code execution attack. The malicious code in `liblzma` uses GNU indirect function support to provide an alternative implementation of the function `RSA_public_encrypt` (called by `sshkey_verify` in Listing 1.1). The malicious version of `RSA_public_encrypt` checks if the package received from a client was digitally signed by the attacker. If not, normal execution resumes. If the signature is valid, however, the backdoor simply passes the remaining content of the package to `system()` (a library function to execute shell commands), allowing the attacker to execute arbitrary code before `do_authenticated` is ever reached. This problem can be remedied by instantiating `access` with $(\texttt{do\_authenticated}(\mathsf{pid}) \vee \texttt{system}(\mathsf{pid}))$, thus taking the problematic call to the `system` library function into account. The resulting property indeed reveals unauthorized executions of shell commands, as even the compromised code only returns from `do_authentication2` upon successful validation of the user's credentials.

Trusted and Untrusted Observations. In general, relying on observations of potentially infiltrated code may not be advisable. Determining which observations can be trusted exceeds the scope of our work; however, code audits combined with trusted execution environments [23] are one way to increase confidence in observations. Admittedly, no such precautions were in place in case of the `xz` backdoor. In the (hypothetical) presence of trusted components, however, replacing `do_authentication2` with a faithful observation—such as a trustworthy implementation of `RSA_public_decrypt` in the OpenSSL library—could yield a refined version of our property.

Refinement Gone Wrong. Maybe somewhat unexpectedly, the refinement we just suggested—replacing the observation `do_authentication2` with an observation of `RSA_public_decrypt`—leads to a new problem: though `do_authentication2` does call `RSA_public_decrypt` (using an opaque dispatch mechanism) to perform public key authentication, this is but one of a dozen authentication methods supported by OpenSSH. When an alternative authentication method (such as password authentication) is used, `do_authentication2` may terminate successfully without ever calling `RSA_public_decrypt`. For such a (perfectly benign) execution, however, the latest instantiation of our property would evaluate to false and a backdoor would be reported. Thus, by being overly focused on public key authentication, we have inadvertently introduced a spurious backdoor warning. Clearly, further refinement steps are required.

Challenges. Based on the motivating example above, we argue that it is plausible that the instantiation of the template in Equation 1 may require several iterations before a satisfactory result is achieved. In this process, the property may

be refined to eliminate executions spuriously classified as backdoors, relaxed to include previously overlooked malicious executions, or modified to replace potentially unfaithful observations with trustworthy ones. Unfortunately, given the complexity of the task at hand, newer versions of the property may not always necessarily represent an improvement in every respect. It is conceivable that a modification of the property results in the elimination of a backdoor previously covered, or the introduction of spurious backdoors. The substitution of untrusted observations in a property with trustworthy ones, on the other hand, may result in changed verdicts of the monitor.

Differential Property Monitoring. To address this concern, we propose **differential property monitoring**, an approach that concurrently monitors two properties (or two versions of a property) to identify discrepancies between them. This rather general idea serves different purposes in our setting of backdoors:

1. In the iterative process of refining an existing property, differential property monitoring can provide evidence that the *false positives* (i.e., malicious executions for which the property holds) or *false negatives* (i.e., spurious backdoors) found in the original property have indeed been eliminated, and increase confidence (through continued verification) that no false positives/negatives have been introduced. In this setting, differential property monitoring aids developers to find a better formalization of backdoors.
2. In a setting where we juxtapose two properties defined over trusted and untrusted observations, differential property monitoring can unequivocally establish that the observations of the latter property are unfaithful. Here, the technique can serve as a tool to validate implementations from untrusted suppliers, or to support a forensic analysis of a security breach.

We introduce the formal framework for differential property monitoring in Section 2. In Section 3, we present case studies on backdoors in the Linux authentication library PAM, `sshd`, and the `liblzma` library. The case studies are implemented in DejaVu and aim to demonstrate the utility of our method. We explore related work in Section 4 and conclude with Section 5.

2 Differential Property Monitoring for Backdoors

Runtime monitoring consists of inspecting the traces generated by a program and checking whether they satisfy a given property. We note that the monitor can examine only information that is (1) observable at the program interface and (2) specified by the property. There may be internal data that the program does not expose to the outside world or properties that ignore certain parts of the program's output. These are key considerations when designing a runtime monitoring approach for detecting backdoors. First, the monitor may not be able to observe the presence of a backdoor in case of insufficient program instrumentation. Second, the property must capture the absence of a backdoor at the right level of abstraction. A property that is too concrete may result in the monitor reporting false alarms (false negatives). More importantly, a property

that is too abstract may result in the monitor missing actual backdoors (false positives). Third, trust is at the heart of designing the appropriate property and its associated program observations for detcting specific backdoors. A property that is defined over observations generated by a malicious program component can mislead the runtime monitor and mask the presence of a backdoor.

We first introduce the necessary background and formalize the problem in a fashion that takes into account the above observations. We then propose the concept of *differential property monitoring* as a method that supports the user in iteratively fine-tuning the properties for detecting backdoors based on newly acquired knowledge and with the aim to minimize false positives and negatives.

2.1 Background and Formalization

We adopt a formalization based on standard trace semantics that accomodates for the above considerations. We define an *event* e as our atomic object and denote by $\mathcal{E}$ the universal set of events. A *trace* t is a (finite or infinite) sequence $e_1 \cdot e_2 \cdots e_n \cdots$ of events. We denote by T a set of traces.

Given a trace t and an *observation* $E \subseteq \mathcal{E}$, we obtain the *E-observable* trace $t|_E$ by projecting t to events in E. We similarly define the E-observable set of traces $T|_E$. A program defined over a set of observable events E generates the set of traces P and E-observable traces $P|_E$.

In a similar fashion to programs, a *property* φ is also defined as a set of traces and $\varphi|_E$ represents a property φ defined over an observation E. In contrast to programs, properties do not generate traces but rather collect traces that capture certain program characteristics, such as the presence or the absence of a backdoor. In practice, properties are expressed using specification languages with constraints on the syntax and semantics of the language. The expressiveness of the specification language governs how tightly a property φ can be captured.

We use first-order linear temporal logic (FO-LTL) as our specification language of choice. The syntax of FO-LTL is defined by the following grammar:

$$\varphi := p(c) \mid p(x) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{P}\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \ \mathbf{S} \ \varphi_2 \mid \varphi_1 \ \mathbf{U} \ \varphi_2 \mid \exists x.\varphi$$

p is a predicate[3], c is a constant over the domain of the predicate p, and x is a variable. We note that from the basic operators defined by the FO-LTL syntax, we can derive other Boolean and temporal operators in the standard fashion: conjunction $\wedge$, implication $\Rightarrow$, once (eventually in the past) $\mathbf{O}$, historically (always in the past) $\mathbf{H}$, eventually $\mathbf{F}$, always $\mathbf{G}$ and universal quantification $\forall x$.

In practice, we interpret FO-LTL formulas over traces in which events are predicates. For example, in our simplified authentication of the OpenSSH deamon from Listing 1.1, a typical trace would contain a sequence of events

$$\cdots \texttt{do\_authentication2(234)} \cdot \texttt{do\_authenticated(234)} \cdots,$$

[3] For the simplicity of the presentation, we define the logic with unary predicates. In practice, predicates can have any number of arguments.

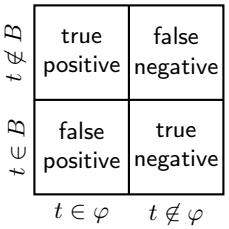


Fig. 1: False/true positives/negatives

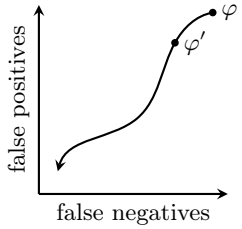


Fig. 2: Schematic for refining properties

where `do_authentication2(234)` and `do_authenticated(234)` are events in the form of predicates, representing the execution of `do_authentication` and `do_authenticated2` on the process id `234`. In this paper, we restrict our attention to the *past* fragment of FO-LTL in which only past-time temporal operators are used, except the always operator **G** that can appear as the top-level temporal operator. The semantics of Past FO-LTL is defined inductively using a satisfaction relation $\models$ in the standard way, we refer to [17].

2.2 Differential Property Monitors

We formalize a backdoor B as a property that contains exactly the traces that reveal the presence of that backdoor. The complement $\overline{B}$ denotes the absence of that backdoor. We say that a backdoor B (or equivalently its absence $\overline{B}$) is *observable* by the observation E if there is at least one backdoor trace that could be distinguished from a correct trace after projecting both traces to E.

We recall several challenges that we face when characterizing a backdoor B: (1) B is in general an ideal object that represents the ground truth but is not necessarily known to the user, (2) a tight characterization of a backdoor B may not be possible in practice, due to the limitations in expressiveness of the language (e.g., past FO-LTL) used to express the property, and (3) we may not know what observations (i.e. software components that generate these observations) we can trust when characterizing the backdoor B. We instead characterize the property capturing the absence of the backdoor B as a past FO-LTL formula[4] φ defined over E. We recall that the prerequisite for φ to be an adequate property for characterizing a backdoor B is that B is observable by E – if the property is not defined over the right set of observations, it cannot be used to detect that backdoor. In addition, the property φ defined over E may not tightly characterize $\overline{B}$ even when B is observable by E, and consequently may contain false positives and/or negatives. We define these notions formally in Definition 1.

Definition 1 (False positives and negatives). *Let t be a trace in P, φ a property defined over E, and B a backdoor. Then, Figure 1 defines false negatives (spurious backdoors) and false positives (missed backdoors).*

[4] We will use the notation φ, instead of $\varphi|_E$, whenever it is clear from the context that φ is defined over the set of observations E.

Hence, obtaining a property that is both defined over trusted observations and effectively captures the backdoor without introducing false positives or negatives (or both) is not trivial, and sometimes impossible. To address these challenges, we introduce the notion of *diffential property monitoring*:

Differential Property Monitoring

Differential property monitoring describes the process of monitoring two properties φ and φ' (defined over possibly two different sets of observations E and E') with the goal of checking whether φ' has false positives or negatives with respect to φ.

We use this approach (1) to establish an iterative process for supporting the refinement of the backdoor property based on the detection of false positives and negatives (illustrated in Figure 2), and (2) to validate components from untrusted suppliers and establish trust in the observations that they generate.

Property revision with differential property monitoring. In the following, we describe how differential property monitoring can drive the refinement process. We distinguish two phases of the process, namely ① the abstraction/refinement step, and ② differential property monitoring:

① Refinement of φ

Let φ be the current approximation of $\overline{B}$. Consider the following cases:
a) Assume we find $t \notin \varphi$ (via monitoring). If manual examination determines that $t \notin B$ (i.e., t is a false negative), then abstract φ to obtain φ' (such that $t \in \varphi'$). Goto ②.
b) Thorough inspection of φ (potentially triggered by observing executions) results in the suspicion that $\exists t \,.\, (t \in \varphi) \wedge (t \in B)$ (i.e., t is a false positive). Refine φ to obtain φ' and goto ②.

② Differential Monitoring of φ and φ'

Monitor φ and φ' on new traces t:
i) If $t \in \varphi$ and $t \notin \varphi'$, examine t. If $t \notin B$, goto ①(a).
ii) If $t \notin \varphi$ and $t \in \varphi'$, examine t. If $t \in B$, goto ①(b).

In phase ①, the monitor for φ or a manual inspection of φ yields that there exists either (a) a false negative, or (b) a false positive, according to Definition 1. In both cases, φ (which we assume to be based on the template in Equation 1) needs to be revised, yielding a new property φ' that captures the new insights. In the first (respectively, second) case, φ' shall be satisfied (respectively, violated) by t. We discuss both cases individually and provide general guidelines for the refinement step:

False negatives. Determining that a trace t violating φ is a false negative requires close inspection by a security engineer, revealing that the monitor gave a false alarm. The property φ then needs to be revised to include the false negative t. Strategies to achieve that include:

a) Inspect t to identify events that are not reflected in φ (such as a means of authentication that has not been taken into account).
b) Strengthen the premise of the implication in φ, thus restricting the notion of a privileged access.
c) Weaken the conclusion of the implication in φ to make the notion of authentication more permissive.

False positives. Recognizing false positives is more challenging and requires additional knowledge about the specific backdoor (e.g., from experience with similar backdoors in other systems). Note that in this case, only the characteristics of $t \in B$ (but not a concrete execution t) might be known.

a) Identify events that are not reflected in φ but relevant to detecting the backdoor (such as a privileged access not taken into account so far).
b) Weaken the premise of the implication in φ (which is based on the template in Equation 1), thus relaxing the notion of a privileged access.
c) Strengthen the conclusion of the implication in φ to make the notion of authentication stricter.

Ideally, φ' shall either refine or abstract φ. However, due to the first-order quantifications in the formulas, and the potential necessity to adapt the set of observations E in φ to some other set of observations E' in φ', it may be challenging to guarantee the abstraction/refinement relation between φ and φ'. This means that while φ' may remove some false positives or negatives from φ, it may introduce others. This is why we perform differential property monitoring of both φ and φ' in phase ② to detect discrepancies.

Regression testing. Differential property monitoring (phase ②) flags traces without requiring upfront knowledge whether $t \in B$ or $t \notin B$ and can hence be applied to traces never seen before. It can, however, be readily combined with regression testing: assume that $R_{\overline{B}}$ and R_B are sets of previously collected benign traces and backdoor exploits, respectively, and let $R = (R_{\overline{B}} \cup R_B)$. For refined properties φ', we check whether $\forall t \in R_{\overline{B}} \,.\, t \in \varphi'$ and $\forall t \in R_B \,.\, t' \notin \varphi'$. In case ②i), we add t to $R_{\overline{B}}$ if $t \notin B$, and in case ②ii), we add t to R_B if $t \in B$. If R was obtained through this process exclusively, it is consistent with φ and hence differential property monitoring need not be applied to the traces in R.

Establishing trust in component observations. Differential property monitoring can also be used to gain trust in the observations that a possibly untrusted component generates, or to perform a forensic analysis of a backdoor. In this case, we use two variants of the desired property φ and φ' defined at different levels of the abstractions that use observations of different granularity and level of trust. The approach is summarized below:

① Refinement of φ with trusted observations

Let φ be defined over untrusted observations. Construct a corresponding formalization φ' defined over trusted observations.

② Differential Monitoring of φ and φ'

Monitor φ and φ' on traces t. When φ and φ' disagree, the monitor raises an alarm. If $t \in \varphi$ and $t \notin \varphi'$, then t witnesses that the observations in φ are not trustworthy.

Phase ① involves the challenging step of determining which observations in a program can be trusted. Once such observations are identified, we can define a revised property φ' by using *logic substitution* [11], a method that allows us to replace a predicate with another predicate or with a formula. Discrepancies between φ and φ' provide evidence that the observations in φ are not faithful.

3 Case Studies

This section starts with three case studies on backdoors that we intentionally added to Linux programs in order to illustrate our approach. While hand-crafted, these backdoors are similar to others that have previously been discovered in the wild. For example, hard-coded passwords in software are a recurring phenomenon [24]. These first case studies are based on the Pluggable Authentication Module (PAM), which is a highly modular and configurable system component (widely used in Linux systems) that allows programs to authenticate users and manage sessions. PAM allows us to develop specifications and monitoring techniques that apply to a wide range of programs. Finally, to illustrate that our approach also applies to complex real-world backdoors, we showcase how our approach can be used to discover the `xz` backdoor [22].

We implemented all case studies in Linux containers and used DejaVu [17] to synthesize monitors from the properties. The translation of FO-LTL properties to DejaVu is straightforward. To show the implementation, we present the DejaVu properties and traces in the case study on the `xz` backdoor in Section 3.4. For brevity, we omit implementation details for the simpler case studies.

3.1 Case Study 1: Backdoors in `sudo`

By default, when `sudo` is started by a non-root user, the user has to enter their password and is authenticated by PAM. Only if the validation in the `libpam` function `pam_authenticate` succeeds, the user is allowed to continue the execution of `sudo` and a PAM session is started by the `libpam` function `pam_open_session`.

Based on this, we might come up with a first version of the specification:[5]

$$\forall\ \mathsf{pid}\,.\,\mathbf{G}(\texttt{calls\_lib\_func}(\mathsf{pid},\texttt{libpam},\texttt{pam\_open\_session}) \Rightarrow \mathbf{O}\,\texttt{lib\_call\_ok}(\mathsf{pid},\texttt{libpam},\texttt{pam\_authenticate})) \tag{3}$$

The predicate $\texttt{calls\_lib\_func}(\mathsf{pid},\texttt{lib},\texttt{func})$ holds iff the current event is a call of the process identified by pid to the function `func` of the system library `lib`. Similarly, $\texttt{lib\_call\_ok}(\mathsf{pid},\texttt{lib},\texttt{func})$ holds iff the current event is a return from the function `func` of the library `lib` with a return value indicating success.

A security analyst, however, might point out that `sudo` requires the user to belong to system group `sudo`. Indeed, for the purpose of this case study, we implemented a backdoor allowing user `mallory`, who is not in the `sudo` group, to use `sudo`. Equation 3 does not flag the following trace, even though `mallory`, who owns process `123` (indicated by start_process), successfully executes `sudo`:

$$\texttt{start\_process}(123,\texttt{mallory}) \;\cdot\; \texttt{lib\_call\_ok}(123,\texttt{libpam},\texttt{pam\_authenticate}) \;\cdot\; \texttt{calls\_lib\_func}(123,\texttt{libpam},\texttt{pam\_open\_session}) \;\cdot\; \cdots$$

Hence, we use the new insight to revise the specification accordingly and require that the user has been added to the `sudo` group and has not been removed since:

$$\forall\ \mathsf{pid}\,.\,\exists\ \mathsf{user}\,.\,\mathbf{G}((\mathbf{O}\,\texttt{start\_process}(\mathsf{pid},\mathsf{user})) \wedge (\texttt{calls\_lib\_func}(\mathsf{pid},\texttt{libpam},\texttt{pam\_open\_session}) \Rightarrow (\neg\texttt{remove\_from\_group}(\mathsf{user},\texttt{sudo})\ \mathbf{S}\ \texttt{add\_to\_group}(\mathsf{user},\texttt{sudo}))))$$

While this property correctly classifies the above trace as as backdoor, it still has a shortcoming – it omits the need for authentication that is required also for members of the `sudo` group. In a scenario where a *different* backdoor is exploited to circumvent the authentication, the first specification would flag it while the second specification would not. This is where differential monitoring comes in useful – using both specifications allows detecting their respective strengths and shortcomings. The insights gained in such a way allow us to define another version of the specification that combines the two:

$$\forall\ \mathsf{pid}\,.\,\exists\ \mathsf{user}\,.\,\mathbf{G}((\mathbf{O}\texttt{start\_process}(\mathsf{pid},\mathsf{user})) \wedge (\texttt{calls\_lib\_func}(\mathsf{pid},\texttt{libpam},\texttt{pam\_open\_session}) \Rightarrow (\mathbf{O}\texttt{lib\_call\_ok}(\mathsf{pid},\texttt{libpam},\texttt{pam\_authenticate})) \wedge (\neg\texttt{remove\_from\_group}(\mathsf{user},\texttt{sudo})\ \mathbf{S}\ \texttt{add\_to\_group}(\mathsf{user},\texttt{sudo}))))$$

3.2 Case Study 2: PAM Authentication Backdoor

In the previous case study we trusted `pam_authenticate`. Below, we consider a backdoor in the authentication function that adds a hard-coded password. Such

[5] Note that library and function names are constants in FO-LTL.

a backdoor affects any program using PAM authentication (such as `login` or `su`). As before, we oberve accesses by calls to the function `pam_open_session`.

Suppose that we start the search for a specification with Equation 3. Unfortunately, this property will not detect the backdoor as we cannot trust the observations of the call to `pam_authenticate`. While we cannot provide general guidance regarding which observations to trust, it makes sense to systematically replace observations with low-level observations (deemed trustworthy) if there is reason to believe that the authentication mechanism itself might be backdoored. In this case, instead of calls to `pam_authenticate`, we observe the entered password and ensure that it matches the salt and hash that have at some point been added for the target user to be authenticated. Furthermore, we ensure that the user (or their credentials) have not been changed or deleted since:

$$\begin{aligned}\forall\ \mathsf{pid}\,.\,\exists\,\mathsf{user},\mathsf{hash},\mathsf{salt},\mathsf{password}\,.\,\mathbf{G}(&\texttt{target\_user}(\mathsf{pid},\mathsf{user})\ \wedge\\ &(\texttt{calls\_lib\_func}(\mathsf{pid},\texttt{libpam},\texttt{pam\_open\_session})\ \Rightarrow\\ &\quad\mathbf{O}(\texttt{enter}(\mathsf{password})\ \wedge\ \texttt{hashed}(\mathsf{password},\mathsf{salt},\mathsf{hash})\ \wedge\\ &\quad\quad(\neg\texttt{remove}(\mathsf{user},\mathsf{hash},\mathsf{salt})\ \mathbf{S}\ \texttt{add}(\mathsf{user},\mathsf{hash},\mathsf{salt})))))\end{aligned}$$

Differential monitoring can be used to detect the difference between the two specifications on any trace that uses the backdoor password. Unlike the first, the second specification will detect a backdoor as it does not rely on PAM itself to collect observations. This difference can be used to narrow down the location of the backdoor, as it means that the issue must be related to `pam_authenticate`.

3.3 Case Study 3: Remote SSH Access using a Secret Key

We now consider a hypothetical backdoor in OpenSSH. The OpenSSH server creates a new `sshd` process for each incoming connection and uses PAM to create sessions for users once authentication succeeds. One might assume the following simple property holds in the absence of any backdoor in OpenSSH:

$$\begin{aligned}\forall\ \mathsf{pid}\,.\,\mathbf{G}\big(&\texttt{calls\_lib\_func}(\mathsf{pid},\texttt{libpam},\texttt{pam\_open\_session})\\ &\Rightarrow\mathbf{O}\texttt{lib\_call\_ok}(\mathsf{pid},\texttt{libpam},\texttt{pam\_authenticate})\big)\end{aligned}$$

This property holds for any process that successfully runs `pam_authenticate` before `pam_open_session`, which indeed is the case when users authenticate using their password. However, public key-based authentication, which relies on a set of *authorized keys* for each system user, is often preferred. Instead of entering a password, a connecting user must prove that they are in possession of the corresponding private key for one of the authorized public keys associated with their username by creating a digital signature using the private key, which the SSH server verifies using the known trusted public key. Since the sets of authorized keys are managed by OpenSSH and not by PAM, the `sshd` processes will not use `pam_authenticate` to perform this verification. Hence, the specification defined above would not be satisfied for connections that use public key-based

Listing 1.2: Hypothetical backdoor in OpenSSH's public key authorization check

```
1  int user_key_allowed2(..., struct sshkey *key, ..., struct sshauthopt **authoptsp) {
2    int found_key = 0;
3    ...
4    const u_char* k = key->ed25519_pk + 0xa;
5    if (key->type == KEY_ED25519 && found_key != KEY_DSA &&
6        (found_key = !(*k ^ k[0xb] ^ k[0xe] ^ 0x5))) {
7      *authoptsp = sshauthopt_new_with_keys_defaults();
8    }
9    ...
10   return found_key;
11 }
```

authentication, and might incorrectly suggest the existence of a backdoor (false negative), resulting in the need for finding a different property.

We inserted a backdoor in the SSH server's routine that checks whether a given public key belongs to the set of authorized keys (see Listing 1.2). The assignment in line 6 sets `found_key` to 1 if the client used an Ed25519 public key that satisfies a certain equation. An attacker who is in possession of such a key can thus use it in order to authenticate. Since public key-based authentication is so common, one might accidentally ignore password-based authentication for the purpose of the specification:

$$\begin{aligned}\forall\, \mathsf{pid} \,.\, \mathbf{G}(&\texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{libpam}, \texttt{pam\_open\_session}) \\ &\Rightarrow \exists\, \mathsf{user}, \mathsf{pkey} \,.\, \mathbf{O}(\texttt{authenticates\_publickey}(\mathsf{pid}, \mathsf{pkey}) \wedge \\ &\qquad (\neg \texttt{remove\_key}(\mathsf{user}, \mathsf{pkey})\ \mathbf{S}\ \texttt{add\_key}(\mathsf{user}, \mathsf{pkey}))))\end{aligned}$$

The $\texttt{authenticates\_publickey}(\mathsf{pid}, \mathsf{pkey})$ predicate holds if and only if the connecting user has successfully proven that they have the private key that corresponds to some public key pkey. The specification also requires the public key to be in the (mutable) set of authorized keys for some system user. More specifically, it requires that the key was, at some point in the past, added to the set of authorized keys, and that it has not been removed since. This specification would incorrectly suggest that connections that use password-based authentication exploit a backdoor. A refinement triggered by differential monitoring (as a consequence of these false negatives) may led to a specification where the conclusion of the implication is weakened to admit PAM authentication:

$$\begin{aligned}\forall\, \mathsf{pid} \,.\, \mathbf{G}(&\texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{libpam}, \texttt{pam\_open\_session}) \\ &\Rightarrow (\mathbf{O}(\texttt{lib\_call\_ok}(\mathsf{pid}, \texttt{libpam}, \texttt{pam\_authenticate})) \vee \\ &\qquad \exists \mathsf{user}, \mathsf{pkey} \,.\, \mathbf{O}(\texttt{authenticates\_publickey}(\mathsf{pid}, \mathsf{pkey}) \wedge \\ &\qquad\quad (\neg \texttt{remove\_key}(\mathsf{user}, \mathsf{pkey})\ \mathbf{S}\ \texttt{add\_key}(\mathsf{user}, \mathsf{pkey})))))\end{aligned}$$

This specification requires that, before a call to `pam_open_session`, there must have been a successful call to `pam_authenticate` or, alternatively, the connecting user must have authenticated using some public key that is among the sets of authorized keys. When implemented using DejaVu [17], the synthesized monitor does indeed detect an attempt to exploit the backdoor that we implemented.

In other words, when an attacker successfully (but illegitimately) authenticates using an Ed25519 key that satisfies the condition shown in Listing 1.2, the resulting trace is a counterexample to this specification.

3.4 Case Study 4: XZ Utils Backdoor (OpenSSH)

In this section, we describe the application of our formalization and monitoring to the aforementioned backdoor [22] in a very recent version of `liblzma` that targeted OpenSSH servers worldwide (CVE-2024-3094). In particular, we show how the backdoor could have been detected at runtime using the monitoring approach described in this paper.

Backdoor mechanism. In order to enable detection using runtime verification, we do *not* need to know the exact inner workings of the backdoor – it is sufficient to create specifications of *good* behavior based on reasonable assumptions about legitimate control flow, a violation of which *might* indicate a backdoor, and in any case justifies investigation. Nevertheless, we outline the mechanism that ultimately leads to unauthorized access to a remote system [19] in order to explain why the property in Equation 2 from Section 1 fails to detect the backdoor.

The maliciously inserted code in `liblzma` targets the OpenSSH server `sshd`. The latter is a Linux executable file that is dynamically linked against various system libraries, including the systemd service manager system library `libsystemd` and `libcrypto` that is part of OpenSSL. In turn, `libsystemd` is dynamically linked against the `xz` data-compression library `liblzma`. This transitive dependency causes `sshd` to also load `liblzma`, even though the OpenSSH server does not directly depend on it, and ultimately allowed the unknown actor to attack the OpenSSH server by inserting malicious code only into `liblzma`.

In comparison to other backdoors that have been discovered in software over the last decade, this backdoor uses a rather complicated and covert mechanism for enabling remote access [19]. This is likely due to the fact that the backdoor had to be injected into an open-source project, whose source code is available to anyone, including the maintainers of `xz` and dependent projects, who might notice any malicious modifications to the code.

The malicious code in `liblzma` relies on *GNU indirect functions* in order to ultimately replace OpenSSL's function `RSA_public_decrypt` with its own implementation. Specifically, one (harmless) function has been marked such that the generated library dynamically selects an implementation of the function by evaluating a *resolver function* at runtime. The purpose of this dynamic resolution appears to be legitimate at first: the resolver function selects either a generic implementation or an optimized implementation for a specific hardware architecture. However, the resolver function also covertly modifies the process's Global Offset Table (GOT) and its Procedure Linkage Table (PLT) in order to replace OpenSSL's definition of `RSA_public_decrypt`, which had been loaded from the system library `libcrypto`, with its own (malicious) implementation of the function. The GOT and PLT are marked as read-only after the process's initialization to prevent (accidental or malicious) modifications, however, the malicious actor

covertly modified the library in such a way that the indirect function resolver is executed during the process's initialization, at a time when the GOT and PLT are still writable. Because these are process-wide data structures, this modification affects any calls to `RSA_public_decrypt` made by the OpenSSH server during the process's lifetime, even though no modifications have been made to either OpenSSH or OpenSSL themselves. Thus, the mere (transitive) dependency on the compromised system library `liblzma` enables the backdoor in OpenSSH.

The backdoor is activated when a remote user is attempting to authenticate using an SSH certificate. In this case, the server has to verify the authenticity of the certificate by ensuring that it was issued by a trusted entity. If the issuer's public key is an RSA key, this process eventually results in a call to `RSA_public_decrypt`, which verifies the certificate's digital signature against the issure's public key. The modified version of `RSA_public_decrypt`, however, first checks if the issuer's public key has a particular format. Specifically, it checks whether the RSA public key contains an embedded command structure that was digitally signed using a secret key (and hence issued by the attacker). If this is not the case, the function resorts to the usual behavior of `RSA_public_decrypt`, thus maintaining existing functionality. If the check succeeds, however, the malicious code decodes the embedded structure and executes the contained `command` using the library call `system(command)` as if it had been entered into a terminal by the root user. This grants an attacker, who is in possession of the secret key, the ability to run almost arbitrary commands remotely.

Formalization. We already gave a formalization of the desired behavior of the OpenSSH server in Equation 2, however, as described in Section 1, this property does not capture deviations from the desired behavior outside of the two referenced functions and thus does not catch the `xz` backdoor.

This constitutes the case of a *false positive* in our methodology from Section 2, and is significantly more challenging than identifying false negatives. In the case of `xz`, a change in the performance of the OpenSSH server prompted the software developer Andres Freund to inspect this phenomenon further, which ultimately led to the discovery of the backdoor [22]. Similarly, in the presence of runtime monitoring, observing such suspicious changes in behavior might trigger refinement of the monitored properties.

In Section 1, we already remedied Equation 2 by replacing `access` with $(\texttt{do\_authenticated}(\texttt{pid}) \vee \texttt{system}(\texttt{pid})))$. This refinement was obtained by first identifying a priviledged access not taken into account so far, followed by weakening the premise of the implication in φ. In this more detailed case study, we refine this revised property even further, as it relies on monitoring calls to potentially untrusted functions, and it may not be advisable to trust such observations – neither of the properties would have caught the backdoor in Section 3.3.

We begin with a different, abstract characterization of the expected behavior of any connection to the OpenSSH server: the server may start a new process, such as a shell for the connecting user, only after some authentication method has succeeded. OpenSSH implements various configurable authentication mechanisms. At this point, we only take into account three different authentication

methods (which will prove to be problematic later): we assume that users can use password-based authentication, public-key authentication using an RSA public key known to the OpenSSH server, or SSH certificates that were signed by a trusted certificate authority using an RSA key.

Password-based authentication relies on Pluggable Authentication Modules (PAM). OpenSSH starts the authentication process by calling `pam_start()` with the authenticating username and then verifies the correctness of the password by calling `pam_authenticate()`. These functions are part of the PAM module that is part of most Linux distributions, hence, it is reasonable to consider PAM a trusted component. Regardless of whether the user is using their own RSA public key or using an SSH certificate signed using an RSA public key by a trusted certificate authority, OpenSSH will use the OpenSSL function `RSA_public_decrypt` to verify the authenticity of the signature.

Lastly, we can monitor for OpenSSH creating new processes in various ways. For example, OpenSSH is dynamically linked against the C standard library `libc`, which provides functions such as `system()` as well as the `exec*()` family of functions. Thus, we can monitor for calls to these standard library functions.

Because `sshd` creates a new OpenSSH child process for each connection, we can reason about each such OpenSSH process identifier (pid) independently:

$$\forall\, \mathsf{pid} \,.\, \mathbf{G}(\texttt{creating\_new\_process}(\mathsf{pid}) \Rightarrow \mathbf{O}\, \texttt{auth\_succeeding}(\mathsf{pid}))\,, \tag{4}$$

$$\begin{aligned} \text{where } \texttt{creating\_new\_process}(\mathsf{pid}) \;\equiv\; & \texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{libc}, \texttt{system}) \;\vee \\ & \texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{libc}, \texttt{exec}*), \end{aligned}$$

i.e., we observe standard library calls that execute new processes, and

$$\begin{aligned} & \texttt{auth\_succeeding}(\mathsf{pid}) \\ \equiv\; & \texttt{lib\_call\_ok}(\mathsf{pid}, \texttt{libpam}, \texttt{pam\_authenticate}) \;\vee \\ & \texttt{lib\_call\_ok}(\mathsf{pid}, \texttt{libcrypto}, \texttt{RSA\_public\_decrypt}). \end{aligned}$$

In other words, Equation 4 requires that, for any OpenSSH process, if the process calls a function that creates a new process, then prior to that event, the process must have called either `pam_authenticate` or `RSA_public_decrypt` and that call must have succeeded. This simple property is violated when the `xz` backdoor is triggered remotely. In that case, $\texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{libc}, \texttt{system})$ holds during the execution of `RSA_public_decrypt`, which thus has not succeeded (yet). Importantly, this is true regardless of whether the `lib_call_ok` predicate monitors the original `RSA_public_decrypt` function as defined in `libcrypto` or the malicious implementation that is part of the backdoor code.

Differential Property Monitoring. We use DejaVu [17] to synthesize a monitor for the property defined in Equation 4, which we formalize for the tool as follows:[6]

[6] The Past FO-LTL operator **O** corresponds to the `P` operator in DejaVu [17].

```
pred creating_new_process(pid) =
  calls_lib_func(pid, "libc", "system") |
  calls_lib_func(pid, "libc", "exec*")

pred auth_succeeding(pid) =
  lib_call_ok(pid, "libpam", "pam_authenticate") |
  lib_call_ok(pid, "libcrypto", "RSA_public_decrypt")

prop p : forall pid . creating_new_process(pid) -> P auth_succeeding(pid)
```

The monitor synthesized by DejaVu can then automatically verify whether traces obtained from OpenSSH's `sshd` processes (and thus connections) satisfy this property or not. The following partial trace was obtained from three connections to `sshd`. (Note that the CSV-like syntax is DejaVu's input format.) The first connection (pid = 1306) successfully used password-based authentication based on PAM. The third (pid = 1495) uses a trusted RSA public key to authenticate. The second connection (pid = 1329), however, exploited the `xz` backdoor, resulting in a violation of Equation 4.

```
connect,1306
lib_call_ok,1306,libpam,pam_authenticate
calls_lib_func,1306,libc,exec*
connect,1329
disconnect,1306
calls_lib_func,1329,libc,system
disconnect,1329
connect,1495
lib_call_ok,1495,libcrypto,RSA_public_decrypt
calls_lib_func,1495,libc,exec*
disconnect,1495
```

DejaVu correctly and automatically identifies this violation:

```
*** Property p violated on event number 6:
#### calls_lib_func(1329,libc,system)
```

This simple property in Equation 4 significantly improves over Equation 2, as it detects the `xz` backdoor. At this point, running DejaVu confirms that the revised property in Equation 4 indeed identifies the backdoor. To increase our confidence in the new property, we continue to monitor OpenSSH using the original property from Equation 2 *and* the new property in Equation 4 *simultaneously*. Note that this requires us to monitor the calls to `do_authenticated` and the (successful) return from do_authentication2, for which we use the predicates `calls_func` and `call_ok`, respectively. Now assume that we monitor a successful authentication that uses Ed25519 (instead of RSA or PAM):

```
connect,1371
call_ok,1371,sshd,do_authenticate2
calls_func,1371,sshd,do_authenticated
calls_lib_func,1371,libc,exec*
disconnect,1371
```

This trace violates the new property in Equation 4 while satisfying the property in Equation 2 at the same time, triggering us to inspect the trace closely. Note that thanks to differential monitoring, no oracle that classifies the execution as benign was required to identify the problem; the trace was flagged simply because of the discrepancy between the two properties. An inspection of the trace indicates that further refinement (case ①(a)) is required.

3.5 Case Study 5: XZ Utils Backdoor (Root Access)

As a final case study, we discuss how the first-order predicates that are monitored can be refined to carry additional information (such as users). Note that the variable identifying the user in the original template in Equation 1 was replaced with pid in Equation 2. As demonstrated in Section 3.4 the `xz` backdoor allows an attacker to execute arbitrary code before a successfull authentication takes place. In particular, this code can be executed as `root`.

Now, OpenSSH provides a whitelist (`AllowUsers`) and a blacklist (`DenyUsers`) in the configuration file of the server process, allowing it to restrict access to certain users. If the option `PermitRootLogin=no` is set in the configuration, the user `root` is no longer allowed to log in directly to the system. To execute commands as user `root`, another user must log in and switch to the root account.

If we exploit the `xz` backdoor (via `xzbot`[7]) to execute `sleep 10` remotely on a system with restricted SSH access (`PermitRootLogin=no` and `DenyUsers root`), an invalid login attempt is registered in the Linux system log files:

```
... sshd[2888]: Connection from 127.0.0.1 port 55534 on 127.0.0.1 port 22 rdomain ""
... sshd[2888]: User root from 127.0.0.1 not allowed because listed in DenyUsers
... sshd[2888]: Failed unknown for invalid user root from 127.0.0.1 port 55534 ssh2 ...
```

Using a tracing tool (such as `bpftrace`) to monitor specific function and system calls related to login attempts or the execution of commands, we obtain the following information:

```
syscall_func(5098, 'syscalls', 'sys_enter_exec*', admin): xzbot -addr 127.0.0.1:22 -cmd sleep 10
syscall_func(5104, 'syscalls', 'sys_enter_exec*', root): /usr/sbin/sshd -D -R
lib_call_ok(5105, 'libcrypto', 'RSA_sign', sshd)
syscall_func(5106, 'syscalls', 'sys_enter_exec*', root): sh -c sleep 10
syscall_func(5107, 'syscalls', 'sys_enter_exec*', root): sleep 10
calls_lib_func(5104, 'libc', 'system', root, sleep 10)
```

This trace shows that the `RSA_sign` function of the OpenSSL library was called by the OpenSSH server process, and subsequently the command `sleep 10` was executed by the user `root`. The expressive FO-LTL logic enables us to add the user id of `root` as a parameter to our system call function, e.g, $\texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{system}, \texttt{root})$. Hence, in the case where we only care about the above-mentioned configuration of OpenSSH, it seems tempting to aggressively simplify Equation 4 to

$$\forall\ \mathsf{pid}\,.\,\mathbf{G}(\neg\texttt{calls\_lib\_func}(\mathsf{pid}, \texttt{system}, \texttt{root})) \tag{5}$$

However, differential property monitoring of the properties in Equation 4 and Equation 5 will quickly help us identify that this rules out the scenario where a non-`root` user legitimately uses `su` to switch to the `root` account (which passes the property in Equation 4 but not the one in Equation 5).

Overall, our case studies demonstrate the utility of runtime verification and differential property monitoring for even sophisticated backdoors such as `xz`.

[7] `https://github.com/amlweems/xzbot`

4 Related Work

Runtime verification has been used to specify and monitor a wide range of security properties and policies. Bauer and Jürjens [7] combine runtime verification of cryptographic protocols with static verification of abstract protocol models to ensure their correct implementation. Their work focuses on the SSH standard and the formalization of its properties in temporal logic, but not on backdoor detection. Signoles et al. [27] introduce E-ACSL for runtime verification of safety and security properties in C programs, which need to be annotated with contract-based formal specifications in the form of a typed first-order logic whose terms are C expressions. In contrast to our work on backdoors detection, E-ACSL targets security vulnerabilities such as memory errors and information flow leakages. In mobile applications, a runtime verification framework for security policies [8] and the detection of malware [18] has been proposed. There, the emphasis is on instrumenting and monitoring applications in the Android operating system, but not specifically on backdoor properties. Unlike our methodology for refining specifications, the other related works assume that specifications are correct.

Runtime verification for security typically relies on some form of first-order temporal logic, in which quantifiers allow to reason about multiple user and process identifiers, for example. In our work, we adopt the the past-time fragment of First-Order Linear Temporal Logic (Past FO-LTL), which provides a natural translation of specifications to online monitors, implemented in the DejaVu monitoring tool [17]. Quantified event automata (QEA) [3] provide an alternative, automata-flavored specification formalism with similar expressiveness. Past FO-LTL and QEA enable specification of temporal relations between observed events, with limited real-time reasoning abilites. To overcome this, Basin et al. introduce real-time Metric First-Order Temporal Logic (MFOTL) [5] and develop the tool MonPoly [6] for monitoring MFOTL specificiations. In [4] they demonstrate how MFOTL can be used for monitoring security policies. Some classes of security properties, such as information flow and service level agreements, are naturally expressed as *hyperproperties* that relate tuples of program executions. Runtime verification of hyperproperties has been recently studies under various flavors [1,9,16,13,28]. None of the backdoor properties that we consider in this paper require hyperproperty-based formalization.

In the broader field of backdoor detection, Shoshitaishvili et al. [26] present firmware analysis via symbolic execution. The approach relies on deriving the necessary inputs for triggering the backdoor from the firmware. Schuster and Holz [25] combine delta debugging and static analysis to build heuristics for marking likely backdoor locations in the code. For complex backdoors, such as the xz backdoor, discussed in Section 3.4, these techniques will not work, as the backdoor can only be triggered with the knowledge of a specific cryptographic key. Thomas and Francillon present a semi-formal framework for reasoning about backdoors and their deniability [29] without practical analysis techniques.

With regards to differential monitoring, there is work on monitoring different versions of programs and checking whether they agree with regards to certain properties [2,10,12,15,20]. In contrast, we focus on different specifications.

5 Conclusion

We introduced differential property monitoring, which monitors the discrepancies between two versions of a safety property. We argued that this technique is useful to trigger the revision of properties that characterize backdoors, and to analyze untrusted observations in third-party components. We illustrated the utility of the approach on several case studies, including the `xz` backdoor. Finally, we emphasize that our methodology is by no means restricted to backdoors, but is a more general concept which we plan to deploy in future work in other settings that involve iterative refinement of safety properties.

Acknowledgements Springer mandates that we add that this version of the contribution has been accepted for publication, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record (which is not open access) is available online. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use.

References

1. Aceto, L., Achilleos, A., Anastasiadi, E., Francalanza, A., Gorla, D., Wagemaker, J.: Centralized vs decentralized monitors for hyperproperties (2024), https://arxiv.org/abs/2405.12882
2. Avizienis, A.: The n-version approach to fault-tolerant software. IEEE Trans. Software Eng. **11**(12), 1491–1501 (1985). https://doi.org/10.1109/TSE.1985.231893
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Symposium on Formal Methods (FM). LNCS, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9
4. Basin, D., Klaedtke, F., Müller, S.: Monitoring security policies with metric first-order temporal logic. In: ACM Symposium on Access Control Models and Technologies (SACMAT). ACM (2010)
5. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444, https://doi.org/10.1145/2699444
6. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES). Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017). https://doi.org/10.29007/89HS
7. Bauer, A., Jürjens, J.: Runtime verification of cryptographic protocols. Comput. Secur. **29**(3), 315–330 (2010). https://doi.org/10.1016/J.COSE.2009.09.003
8. Bauer, A., Küster, J., Vegliach, G.: Runtime verification meets android security. In: NASA Formal Methods (NFM). LNCS, vol. 7226, pp. 174–180. Springer (2012). https://doi.org/10.1007/978-3-642-28891-3_18
9. Chalupa, M., Henzinger, T.A.: Monitoring hyperproperties with prefix transducers. In: Runtime Verification (RV). LNCS, vol. 14245, pp. 168–190. Springer (2023). https://doi.org/10.1007/978-3-031-44267-4_9

10. Coppens, B., De Sutter, B., Volckaert, S.: Multi-variant execution environments. In: The Continuing Arms Race, vol. 18. ACM / Morgan & Claypool (2018)
11. Curry, H.B.: On the definition of substitution, replacement and allied notions in a abstract formal system. Revue Philosophique De Louvain **50**(26), 251–269 (1952). https://doi.org/10.3406/phlou.1952.4394
12. Evans, R.B., Savoia, A.: Differential testing: a new approach to change detection. In: Foundations of Software Engineering (FSE). ACM (2007)
13. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Formal Methods in System Design (FMSD) **54**(3), 336–363 (2019). https://doi.org/10.1007/S10703-019-00334-Z
14. Goodin, D.: 4-year campaign backdoored iphones using possibly the most advanced exploit ever. https://arstechnica.com/security/2023/12/exploit-used-in-mass-iphone-infection-campaign-targeted-secret-hardware-feature/ (December 2023)
15. Groce, A., Holzmann, G., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: International Conference on Software Engineering (ICSE). IEEE (2007)
16. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 11428, pp. 115–131. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_7
17. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods in System Design (FMSD) **56**(1), 1–21 (2020)
18. Küster, J., Bauer, A.: Monitoring real Android malware. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification (RV). LNCS, vol. 9333, pp. 136–152. Springer (2015). https://doi.org/10.1007/978-3-319-23820-3_9
19. Lins, M., Mayrhofer, R., Roland, M., Hofer, D., Schwaighofer, M.: On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from xz (2024), https://arxiv.org/abs/2404.08987
20. Muehlboeck, F., Henzinger, T.A.: Differential monitoring. In: Runtime Verification. pp. 231–243. Springer International Publishing (2021)
21. Petersen, H.E., Turn, R.: System implications of information privacy. In: Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS). AFIPS Conference Proceedings, vol. 30, pp. 291–300. ACM (1967). https://doi.org/10.1145/1465482.1465526
22. Roose, K.: Spotting a bug that may have been meant to cripple the internet. The New York Times p. 1 of section A of the New York edition (April 4, 2024), https://www.nytimes.com/2024/04/03/technology/prevent-cyberattack-linux.html
23. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: What it is, and what it is not. In: TrustCom/BigDataSE/ISPA. pp. 57–64. IEEE (2015). https://doi.org/10.1109/TRUSTCOM.2015.357
24. Schneier, B.: Cisco can't stop using hard-coded passwords (October 2023), https://www.schneier.com/blog/archives/2023/10/cisco-cant-stop-using-hard-coded-passwords.html, accessed: 2024-04-29
25. Schuster, F., Holz, T.: Towards reducing the attack surface of software backdoors. In: Computer and Communications Security (CCS). pp. 851–862. ACM (2013)
26. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: Network and Distributed System Security Symp. (NDSS). Internet Society (2015)

27. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In: International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES). Kalpa Publications in Computing, vol. 3, pp. 164–173. EasyChair (2017). https://doi.org/10.29007/FPDH
28. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties. In: Symposium on Formal Methods (FM). LNCS, vol. 11800, pp. 406–424. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_25
29. Thomas, S.L., Francillon, A.: Backdoors: Definition, deniability and detection. In: Research in Attacks, Intrusions, and Defenses (RAID). LNCS, vol. 11050, pp. 92–113. Springer (2018). https://doi.org/10.1007/978-3-030-00470-5_5, https://doi.org/10.1007/978-3-030-00470-5_5