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).

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

```c
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 $\mathbf{O}$ 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 $(\mathtt{do\_authenticated}(\mathsf{pid}) \vee \mathtt{system}(\mathsf{pid}))$, 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.

	$t \in \varphi$	$t \notin \varphi$
$t \notin B$	true positive	false negative
$t \in B$	false positive	true negative

Fig. 1: False/true positives/negatives

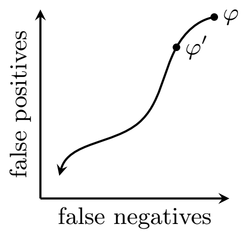

Fig. 2: Schematic for refining properties

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 $\mathbf{G}$ 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`:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

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

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

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