Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications

Kevin Milner*, Cas Cremers*, Jiangshan Yu†, and Mark Ryan‡

March 8, 2017

*University of Oxford, United Kingdom
†University of Luxembourg, Luxembourg
‡University of Birmingham, United Kingdom

Abstract

We develop foundations and several constructions for security protocols that can automatically detect, without false positives, if a secret (such as a key or password) has been misused. Such constructions can be used, e.g., to automatically shut down compromised services, or to automatically revoke misused secrets to minimize the effects of compromise. Our threat model includes malicious agents, (temporarily or permanently) compromised agents, and clones.

Previous works have studied domain-specific partial solutions to this problem. For example, Google's Certificate Transparency aims to provide infrastructure to detect the misuse of a certificate authority's signing key, logs have been used for detecting endpoint compromise, and protocols have been proposed to detect cloned RFID/smart cards. Contrary to these existing approaches, for which the designs are interwoven with domain-specific considerations and which usually do not enable fully automatic response (i.e., they need human assessment), our approach shows where automatic action is possible. Our results unify, provide design rationales, and suggest improvements for the existing domain-specific solutions.

Based on our analysis, we construct several mechanisms for the detection of misuse. Our mechanisms enable automatic response, such as revoking keys or shutting down services, thereby substantially limiting the impact of a compromise.

In several case studies, we show how our mechanisms can be used to substantially increase the security guarantees of a wide range of systems, such as web logins, payment systems, or electronic door locks. For example, we propose and formally verify an improved version of Cloudflare's Keyless SSL protocol that enables key misuse detection.

1 Introduction

Most secure systems depend on secrets, and in particular cryptographic keys. Consequently, many technical and procedural measures have been developed to prevent the leakage of secrets, such as hardware security modules.

In reality, secrets are often compromised in various ways, either through compromising a system holding them, implementation bugs, or cryptanalysis. This has driven the need to design mechanisms to cope with the compromise of a secret, such as key revocation procedures, user blacklisting, or disabling the relevant services entirely. However, independently of designing these response mechanisms, a core question remains: how can we tell if a secret has been compromised? In other words: *when* are we supposed to invoke these response mechanisms?

If an attacker compromises a secret but never makes any visible use of it, it can be hard (or even impossible) to detect the compromise. However, in many cases, the attacker has some other goal, which it can only perform using the secret. For example, to log into a service, to request a document, or to trigger a specific action of the system like opening a door.

This observation is used by mechanisms like SSH's reporting of the last login, or Gmail's reports of current sessions. In these settings, the service informs the user about the details of their prior session(s). If an attacker compromises the user's secret and logs in, the user could, in theory, detect this

manually upon their next login. In practice, users often ignore this information or cannot be expected to remember precisely when they logged in to each service they use.

Further mechanisms that aim to facilitate detection include Certificate Transparency and its relatives, which aim to make relevant uses of certificate authority (CA) publicly observable, thereby making it possible to detect misuse. However, while these mechanisms typically provide a means to observe key uses, they do not prescribe how to determine if the observed key use is honest or when to invoke a response mechanism if it is not. In practice, a domain owner or CA must manually check for an inappropriately issued certificate in the log, and then decide to take action—which may involve further out-of-band communication to obtain additional details not visible in the log—before any response mechanism is invoked.

This leads to several questions. First, is it possible to *automatically* determine that a secret is being misused at the protocol layer, to avoid reliance on human input? In this case, what guarantees could be given? In particular, we focus on detection mechanisms that do not yield false positives, which enables a positive detection to automatically trigger a response mechanism that is appropriate for the secret involved (such as key revocation).

Second, what are the underlying observations that make such mechanisms work? Is there any connection between the various mechanisms that aim to detect the misuse of secrets? What are the limits of detection, and what principles would be useful to protocol designers, in the style of [1]?

Contributions. Our main contributions are the following:

First, we provide the first general foundations for *provably sound* detection of the misuse of secrets, i.e. detection that allows for automatic response. Our focus on detection as a verifiable security property without false positives leads to solutions that can be used to automatically revoke keys, access, or invoke other countermeasures. Our foundational approach also provides new insights into the design choices in existing mechanisms. For example, for detecting the misuse of a Certificate Authority's key on the internet, our results show that both the domain owner and the CA could automatically perform a certain kind of detection ("acausal detection") that no other parties can perform, which leads to suggestions for improving detection mechanisms in this domain, such as Certificate Transparency. More generally, our results reveal which agents can perform which types of detection automatically, clearly delineating what is possible and impossible to achieve in theory.

Second, we apply our foundational work to identify and develop several principles and generic protocol constructions to automatically detect the misuse of secrets. We then show how such constructions can be applied to improve the security of a variety of existing mechanisms, ranging from the previously mentioned certificate authorities to card-based door access. For example, we propose a simple modification to Cloudflare's Keyless SSL protocol [10] to enable the customer to detect misuse of the CDN's keys, which can directly trigger revocation. We formally verify our proposals using the Tamarin prover. We additionally use our techniques to suggest improvements to the Common Access Card [23], and the certificate creation procedures of CAs.

We proceed as follows. In Section 2, we provide an informal introduction to the idea of misuse detection. We then, in Section 3, develop foundations and protocols for the automatic detection of the misuse of secrets. We construct example protocols and apply these constructions to concrete application examples in Section 4. We describe related work in Section 5 and conclude in Section 6.

2 Foundations: an informal introduction

In this work we investigate a problem which has only been studied in limited instances so far: the automatic detection of the misuse of secrets. In an ideal world it would be possible to indefinitely prevent secrets from being compromised. Realistically, we cannot assume this is guaranteed, which drives the need for mechanisms and procedures that can mitigate the damage of a compromised secret.

We observe that if an attacker silently obtains a secret but performs no visible actions based on this information, the compromise fundamentally cannot be detected. Furthermore, if the attacker obtains all necessary secrets to impersonate the original owner, performs actions using those secrets that are identical to the expected behaviour of the original owner, and the original owner performs no further actions (e.g., because they are deceased), then to all other participants the attacker's behaviour is indistinguishable from the original owner. In a way, the attacker would have completely taken over the life of the original owner. Thus, informally, the only situation in which we can hope to detect the misuse of those secrets is when the attacker deviates—or rather, is forced to deviate—from the original owner's behaviour or ongoing actions.

In this work, we are interested in protocols in which participants obtain specific evidence of deviation (that is, there are no false positives). This would allow detecting agents to immediately trigger appropriate countermeasures, such as disabling a service, revoking keys, or blacklisting users. Thus, our work contrasts with the field of *anomaly detection*, where one of the challenges is to detect behaviours that are allowed by the specification, but unlikely to occur during normal usage; such detection is typically plagued by false positives, and it is hard to take countermeasures as a result.

Consider the following examples of protocols which allow agents to differentiate adversary action from action by the honest agents, which each examine a different aspect of detection that we will return to in Section 3.

Example 1. *Alice has a secret k_A which she can use to authenticate messages. The adversary compromises this secret, and sends an authenticated message which is obviously incorrect. For example, the authenticated message might be "I compromised this secret".*

Example 1 is unlikely to occur in practice, but it is still a valid action the attacker could take so it is important to take it into account.

Example 2. *Alice and Bob have signing keys k_A and k_B respectively, and send each other messages authenticated with their keys over a public channel. They each maintain a counter, and when Alice sends a message to Bob she increments her counter, generates a new nonce n_A and includes them both in her message along with the last nonce received from Bob. Upon receiving this message Bob checks that his last nonce matches, increments his counter, and checks that it matches the one in the message. Similarly, when Bob sends a message to Alice, he includes a newly generated nonce n_B, his counter value, and Alice's last nonce n_A. The next message from Alice contains a new nonce n'_A, n_B, and an incremented counter value, the next message from Bob a nonce n'_B, n'_A, and his counter value, etc.*

Example 2 illustrates a simple case in which misuse can be detected. If an attacker gains knowledge of k_A and the current value of the counter, and injects a new message purporting to be from Alice, then Alice's and Bob's value of the counter will become de-synchronised and they could detect upon comparison that k_A was misused. However, this is somewhat limited, as an attacker with knowledge of both keys who observes a counter value could strike up conversations with Bob, then wait for Alice to send messages. By intercepting these and returning a message to Alice which appears to be from Bob the adversary can increment Alice's counter until it matches, and then inject one more message to each to resynchronize their nonces. Alice and Bob are left in a state as if the attacker were never involved.

Note that because Alice and Bob's counter values rely only on the number of messages exchanged and not on their content, it is impossible to determine if they agreed on all previous message content. Thus, the attacker can resynchronize them even after they have disagreed about the messages exchanged.

Example 3. *Instead of using a counter, Alice and Bob adopt a system of 'rolling nonces with hash chains'. When Alice sends her authenticated message to Bob, she includes a new nonce n_A and a hash chain of the previous nonces used by both parties in the conversation. Bob then checks the value of the hash chain matches his own, and when sending a message to Alice does likewise, including a new nonce n_B and extending the hash chain with n_A. The next message from Alice contains a new nonce n'_A and the hash chain extended with n_B, etc.*

In Example 3, suppose an attacker obtains Alice's key k_A along with the current nonce and hash chain. The attacker can inject conversations with Bob, which necessarily extends Bob's hash chain with new values. If the attacker ever stops intercepting messages between the two, his session will be detected, since the hash chain of Alice will not match and the adversary has no way to 'rewind' Bob's additions to his hash chain. Indeed, even if both keys k_A and k_B are compromised, this example with hash chains allows for detection if ever the attacker tries to back out of the conversation, as any session the attacker carries out with either of them has an irreversible effect on their state.

3 Foundations and design space

In this section, we develop formal foundations and explore the design space for detecting of secret misuse. While our contributions can be informally understood and applied in practice by skipping most of this section and immediately moving to Section 3.5, our formal work serves the following purposes: it enables us to precisely define the relevant concepts, explore the design space more systematically, and will enable us later to *prove* that some protocols indeed achieve detection. We will use the resulting

definitions in Section 4 to develop concrete protocols, prove their correctness, and show how to improve existing systems.

First, in Sections 3.1 to 3.3, we build the necessary framework to formally define what it means to soundly detect compromise, and what is necessary for detection. This leads us to classify the possible ways misuse can be observed into three broad categories in Section 3.4 and show that they together form a complete categorization. Finally, we combine these elements for the design space in Section 3.5.

3.1 Basic mechanisms

We introduce basic notation for a generic class of protocols and an abstract notion of detection. This enables us to formally define what it means to (soundly) detect compromise, and what is necessary for detection. We then isolate in Section 3.4 three different ways agents can observe key misuses: inconsistency with a protocol specification, contradictions, and acausality. We will use these three types of observation to guarantee detection in particular scenarios, and apply this in Section 4 to design and improve protocols.

We assume a finite set of agents Agent as participants, each of which has some associated state, access to a random number generator, and which can communicate only through sending and receiving messages on a network. Agents perform actions according to a *protocol*. A protocol is a deterministic algorithm to be run on a Turing machine with agent state as input, which returns an action to perform. Such actions may include sampling the random number generator, sending or receiving messages on the network, modifying their state, etc. We write $\mathrm{Protocol}$ to denote the set of all protocols.

To model adversarial activity, we assume the existence of an adversary with similar resources to the agents, but with the additional ability to perform actions which remove messages from the network and compromise parts of agent state. Adversary actions are provided by a deterministic algorithm, which we call an *adversary model*. It runs on a Turing machine, taking adversary state as input and outputting an action for the adversary to perform. We denote the set of all adversary models Adv.

The definition above does not allow for malicious agent activity, since all agents are assumed to follow the protocol. We emulate malicious agents instead through the adversary model, which may include completely compromising the state of some agents. Since agent actions are a function of their state, and since all communication with other agents occurs through the adversary-controlled network, this is sufficient to allow adversary emulation of an agent. This makes it easier to abstractly distinguish potentially malicious actions from honest and correct actions in the trace, while allowing for over-approximation of the abilities of malicious agents (since the adversary model may include controlling the network or compromising additional agents).

Each combination of a protocol $P \in \mathrm{Protocol}$ and adversary model $\mathcal{A} \in \mathrm{Adv}$ gives rise to a transition system with agent states, the network as a set of messages, and the state of the adversary. We log each action performed by agents or the adversary as events in a trace, with the requirement that the trace contains sufficient information to reconstruct the state of the adversary and every agent at the end of the trace. For example, an agent performing an action to sample the random number generator, would be logged as an event including the resulting value.

Generally we do not care about the specific actions performed by the agents or the adversary, or their resulting encoding in the trace, other than requiring an abstract way to refer to certain events relevant to detection. This allows us to restrict the actions of participants as little as possible while still having well defined communication structure. We use $\texttt{compromise}(k)$ to refer to any adversary event that compromised some data k from any agent's state. We use `send` and `recv` to refer to any agent event which sent (resp. received) on the network, parametrized by the message involved. Finally, we denote detection of a compromised k by a special agent event $\texttt{detect}(k)$.

The initial state of the agents includes both agent-specific data as well as any public data assumed to be known to the adversary as well as the agent (e.g. some settings may assume a public key infrastructure). Adversary initial state contains only this public data. Since we do not bound the computation time of the agents or adversary, we instead assume a symbolic model of security in which a term algebra (e.g. that of TAMARIN [19]) defines how terms may be derived.

In order to discuss a particular subsequence of events in a trace, we define the sequence projection operator. For a set S of trace events, the projection $|_S$ is defined as

$$\langle\rangle|_S = \langle\rangle$$
$$(\langle e\rangle \cdot tr')\,|_S = \begin{cases} tr'|_S & \text{if } e \notin S \\ \langle e\rangle \cdot (tr'|_S) & \text{if } e \in S. \end{cases}$$

We define the projection $\restriction_S$ as the projection onto the complement of S, $|_{S^c}$. For shorthand, we enumerate some common projections that we will use throughout this section:

- For a set $X \subseteq \text{Agent}$, $|_X$ for all trace events e such that one of the agents in X is performing e,
- $|_{c(k)}$ for all $\texttt{compromise}(k)$ events,
- and $|_{\texttt{event}}$ for all trace events e of type `event`.

Projection is distributive over sets of sequences, so a projection of a set of sequences is the set of each sequence with the projection applied. We address elements of a sequence s as $s_1 \dots s_{|s|}$ from the first to the last element. We overload set notation for sequences and write $e \in s$ for a sequence s if and only if $\exists i \,.\, s_i = e$.

We focus on detection protocols that can automatically trigger an appropriate response when they detect, such as key revocation, disabling services, or blacklisting users. To enable this, it is important that there are no false positives. Formally,

Definition 4 (Soundly detecting protocol)**.** *We say a detecting protocol $P \in \text{Protocol}$ is a sound for an adversary model $\mathcal{A} \in \text{Adv}$ if*

$$\texttt{sound}(P, \mathcal{A}) \equiv \forall tr \,.\, tr \in Tr(P, \mathcal{A}) \implies \forall k \,.\, \big(\texttt{detect}(k) \in tr \implies \texttt{compromise}(k) \in tr\big).$$

In this paper we do not prescribe a response mechanism for key compromise, since this is an orthogonal area of research (and often involves side-channels or other scenario- or system-specific resources). We instead discuss which parties can detect and when. Soundness enables any detecting party to immediately trigger whichever response mechanism it deems appropriate.

3.2 Reasoning about agents

In order to reason about agent capabilities, we must be able to talk about the their state as well as the possible actions they can perform under particular constraints. We begin with some notation to discuss the state of agents after a trace. Since trace events are, by definition, enough to determine how agent state changes with each action, the state of some agents at some time along with a sequence s of events are sufficient to determine the state of those agents after s. This is formally stated in Corollary 6.

Definition 5 (State after a trace)**.** *Let P be a protocol and let $X \subseteq \text{Agent}$ be a set of agents. We introduce the notation state(tr, X) to represent the collective state of the agents of X after a trace tr.*

Corollary 6 (State convergence)**.** *Let tr and tr' be two traces in $T = Tr(P, \mathcal{A})$ such that state$(tr, X) =$ state(tr', X), and $X \subseteq \text{Agent}$. Then*

$$\forall s \,.\, tr \cdot s \in T \wedge tr' \cdot s \in T \implies \textit{state}(tr \cdot s, X) = \textit{state}(tr' \cdot s, X).$$

The state of particular agents' after trace tr are, by necessity, some function of $tr|_X$, since all state changes of an agent arise from actions they perform. Notably, this requires that state$(tr, X) =$ state(tr', X) if $tr|_X = tr'|_X$.

State convergence is a particularly useful property, because it implies that a subset of agents cannot differentiate two traces in which their combined states are the same, unless they later receive a message that is only possible in one of the two. In fact, we can lift this to prove practical limitations on when it is possible to detect even when agents can run an arbitrary protocol between themselves. We define *protocol extensions* to capture the events that could occur running a secondary protocol, without an adversary, after a particular trace.

Definition 7 (Protocol extension)**.** *Let $T = Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$. A protocol extension performed by a set $X \subseteq \text{Agent}$ beginning from the trace tr is the set of all sequences of agent events s performed by agents in X such that $tr \cdot s \in T$, and s is independent of all prior network events. Formally,*

$$SP(tr, T, X) \equiv \{s \mid (tr \cdot s) \in T \wedge (s|_X = s) \wedge \forall m, i \,.\, \big(s_i = \texttt{recv}(m) \implies \exists j < i \,.\, s_j = \texttt{send}(m)\big)\big\}.$$

We use $SP(tr, T)$ as shorthand for $SP(tr, T, \text{Agent})$, which is equivalent to omitting only adversary events from the protocol extensions.

Intuitively, these protocol extensions represent what a set of agents can do by running a protocol amongst themselves after a particular trace, in an ideal environment where no adversary interferes. State convergence can be leveraged to show a useful property of the protocol extensions across all possible protocols.

Lemma 8. *Let $T = Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$, and $X \subseteq \text{Agent}$ a set of agents. Then*

$$\forall tr, tr' \in T \;.\; (\mathit{state}(tr, X) = \mathit{state}(tr', X)) \implies SP(tr, T, X) = SP(tr', T, X).$$

That is, for every protocol, any two traces tr and tr' where $\mathit{state}(tr, X) = \mathit{state}(tr', X)$ have the same X-protocol extensions.

Proof. Assume otherwise; that is, there is a trace suffix s in $SP(tr, T, X)$ that is not in $SP(tr', T, X)$.

If $s \notin SP(tr', T, X)$, then by the definition of protocol extensions either $s|_X \neq s$, there are `recv` events with no corresponding `send` in s, or $tr' \cdot s \notin T$. The first two are trivially false by the requirement that $s \in SP(tr, T, X)$, so it must be true that $tr' \cdot s \notin T$. We will construct $tr' \cdot s$ recursively to show that this is false.

Take the first element of s, which we will call e such that $s = \langle e \rangle \cdot s'$ for some sequence s'. The set T is prefix-closed since it is generated by a protocol, and by the definition of protocol extension, $tr \cdot s \in T$, so $tr \cdot \langle e \rangle \in T$.

If $tr' \cdot \langle e \rangle \notin T$, then this must be because the action corresponding to the event e cannot be performed after tr'. The event can only rely on receivable messages on the network, the output of the random number generator, or the state of the agent, so one of these must differ between tr and tr'. However, the antecedent requires that both the states and sent messages are identical, and the random number generator does not depend on the prior trace, so none of these can be the case and thus $tr' \cdot \langle e \rangle \in T$.

By Corollary 6, $\text{state}(tr \cdot \langle e \rangle, X) = \text{state}(tr' \cdot \langle e \rangle, X)$, and because every trace is finite we are left with a shorter s' on which we can repeat the argument above to eventually find that $tr' \cdot s \in T$, a contradiction. □

Lemma 8 allows us to begin reasoning about the space of possible actions a set of agents can take. It shows that after a trace, a set of agents performing any protocol at all amongst themselves are still limited to some computation over their collective state.

Note there is an equivalent definition of soundness in terms of protocol extensions.

Definition 9 (Soundly detecting)**.** *A detection protocol $P \in \text{Protocol}$ is sound for adversary model $\mathcal{A} \in \text{Adv}$ if*

$$\mathsf{sound}(P, \mathcal{A}) \equiv \forall tr, s \;.\; tr \in Tr(P, \mathcal{A}) \wedge s \in SP(tr, T) \implies \forall k \;.\; (\mathtt{detect}(k) \in s \implies \mathtt{compromise}(k) \in tr)\,.$$

3.3 Observability of misuse

Whether a usage of a key is 'correct' in general may not be possible to determine from the limited perspective of an agent. To detect misbehaviour, and its subsequent attribution to the misuse of a secret, the protocol (or in a wider sense, the security mechanism) must be designed to make the misuse observable by the agent in question. We first give two examples to provide intuition about the type of designs that (fail to) accomplish this, before providing a more formal treatment of observable misuse to build useful detection protocols.

Ideally, it would be possible to soundly detect any compromise by the adversary. There is however an upper bound on how much can be detected: intuitively, there is no possible protocol for a set of agents to soundly detect secret misuse if that misuse had no effect on them. We formalize this below, using the protocol extension properties discussed above.

Lemma 10 (Sound detection requires state)**.** *For a trace $tr \in Tr(P, \mathcal{A})$ generated by a protocol P with adversary $\mathcal{A}$, a secret k, and a set X of agents,*

$$\forall s, tr' \;.\; s \in SP(tr, Tr(P, \mathcal{A}), X) \wedge \mathtt{detect}(k) \in s \wedge tr' \in Tr(P, \mathcal{A}) \wedge \left(tr'|_{c(k)} = \langle\rangle\right) \wedge \mathit{state}(tr, X) = \mathit{state}(tr', X) \implies \neg\mathsf{sound}(P, \mathcal{A}).$$

That is, if a set X of agents detect the misuse of k in a trace $tr \in Tr(P, \mathcal{A})$ when that state could also be reached in a trace without compromise, then the protocol cannot be sound.

Proof. Assume it is possible for the agents in X to soundly detect after tr, and thus a suffix $s \in SP(tr, Tr(P, \mathcal{A}), X)$ where $\mathtt{detect}(k) \in s$.

The antecedent requires a trace tr' where

$$tr' \in Tr(P, \mathcal{A}) \wedge \left(tr'|_{c(k)} = \langle\rangle\right) \wedge (\text{state}(tr, X) = \text{state}(tr', X)),$$

and from Lemma 8,

$$SP(tr, Tr(P, \mathcal{A}), X) \subseteq SP(tr', Tr(P, \mathcal{A}), X).$$

Thus $s \in SP(tr', Tr(P, \mathcal{A}), X)$. Since the agents detect in s after a trace with no compromise events, the detection cannot be sound. □

The requirement that the state of the agents could arise in a restricted trace set (in this case, traces with no compromise of k) is a useful one, which we formalize in terms of agent state being *consistent* with a trace set.

Definition 11 (State consistent with a trace set)**.** *The state of some agents $X \subseteq \mathrm{Agent}$ after a trace tr is consistent with a trace set T if there is at least one trace in T which leaves X in the same state as tr.*

$$\mathsf{consistent}(X, tr, T) \equiv \exists tr' \in T \,.\, (\mathit{state}(tr, X) = \mathit{state}(tr', X)) \,.$$

We say that misuse of a secret k in a trace $tr \in Tr(P, \mathcal{A})$ is *unobservable* by a set X of agents when

$$\mathsf{consistent}(X, tr, \{t \mid t \in T \wedge t\!\upharpoonright_{c(k)} \in T\}).$$

A trace involving key misuse which leaves the agents in some state that is also reachable without an compromise of the key limits the ability of the agents in X to detect; by Lemma 10 there is no idealized protocol the agents of X could run to detect the misuse.

It is important to note that, while observability of secret misuse is necessary for a set of agents to soundly detect it, it is not sufficient to guarantee that deciding whether to detect can be done tractably (i.e. in a polynomial amount of time). For example, consider a toy protocol where an agent generates a random value with some property and sends the output of a one-way permutation applied to that value over the network signed with their key—detecting misuse of that key may require inverting the permutation to check if the input value had the correct property.

3.4 Categorizing observable misuse

Lemma 10 shows that a set $X \subseteq \mathrm{Agent}$ must reach a collective state inconsistent with the set of all traces without compromise of k to have a possibility of soundly detecting it. In this section we show a categorization of different ways an inconsistent state might be reached, and prove some properties of them which should be considered when designing or modifying a protocol to detect secret misuse.

We divide the ways of observing misuse into three categories, based on the messages received by an agent. The first, *trace-independent inconsistency* refers to a received message that could not have occurred at all without compromise. The second, an *observation of contradiction*, refers to the observation of a sequence of messages which, while each individually possible, could not occur in that sequence without compromise. Finally, an *observation of acausality* is when a sequence of received messages requires action on the part of an agent in order to occur in a trace set, but has occurred without such an action. This final type of observation requires agents to be in a position where they would know if the action did not occur.

3.4.1 Trace-independent inconsistency

The simplest way in which agents can determine that the current trace is inconsistent with a trace set is by receiving a message which could not occur in any trace of that trace set. This category of misuse event is observable 'statelessly' in the sense that it is inconsistent with the trace set independently of the current trace. As such, we refer to this category of observability as *trace-independent inconsistency*.

We formalize it as the negation of the predicate spec, representing the ability to receive a message in a trace set, where

$$\mathsf{spec}(m, T) \equiv \exists tr \in T \,.\, \mathtt{recv}(m) \in tr.$$

and messages which cannot be received in a trace set are referred to as *out-of-specification*. The latter are not expected to arise often. Example 1 in Section 2 illustrates this.

3.4.2 Observing contradictions

The messages in a trace are contradictory compared to a trace set if each message can occur individually but the sequence cannot occur in any trace of the trace set. This is formalized with the predicate contra.

Definition 12 (Contradictory messages)**.** *Given a set $X \in \mathrm{Agent}$, a trace tr, and a trace set T, we say that the agents of X have received a contradictory sequence of messages when*

$$\mathsf{contra}(X, tr, T) \equiv \big(\forall m \,.\, \mathtt{recv}(m) \in tr \implies \mathsf{spec}(m, T)\big) \wedge \big(\forall tr' \in T \,.\, tr|_X|_{\mathtt{recv}} \neq tr'|_X|_{\mathtt{recv}}\big).$$

Example 2 in Section 2 can detect because of the observability of contradictory messages. In that example, each message received from the other party is expected to include the next counter value, so an agent could detect if they saw two messages with the same counter value even if each message would be valid on its own.

A stronger example making use of contradictory messages to detect is found in transparency overlays like Certificate Transparency, where the public log produces signed tree heads for the auditors. These signed tree heads are expected to be mutually consistent, and misuse of the log server's key could be detected in principle by receiving a tree head which is not consistent with another tree head produced with the key.

Both trace-independent inconsistency and contradictory messages allow an agent to store evidence of key misuse, since the message or messages involved are enough to detect irrespective of the receiving agent's state. This allows, in transparency overlays for example, the misuse of a log server's key to be proven to third parties by showing them two inconsistent signed tree heads.

3.4.3 Observing acausality

While the previous two categories reason about received messages, it is also possible to detect based on agent state directly by counterfactual reasoning. For example, an agent storing all prior uses of their key can identify misuse of their key if they see a usage that is not in their state. This extends in more complex ways: transparency overlays are based on the ability of an identity or domain owner to notice an entry in the log that they did not request, on the assumption that only the owner should be initiating the process to add an entry to the log.

We define a notion of *causal precedence*, where an agent causally preceding the messages of a trace has some guarantee that they are required every time some sequence of messages occurs in the trace set.

Definition 13 (Causal precedence)**.** *A set of agents X* causally precedes *the messages of a trace tr in a trace set T if there is some trace in which those messages can be received, and in every such trace X must have participated by sending at least one message. Formally,*

$$\mathsf{causal}(X, tr, T) \equiv \big(\exists tr' \in T \,.\, tr|_X|_{\mathtt{recv}} = tr'|_X|_{\mathtt{recv}}\big) \wedge \big(\forall tr' \in T \,.\, tr|_X|_{\mathtt{recv}} = tr'|_X|_{\mathtt{recv}} \implies tr'|_X|_{\mathtt{send}} \neq \langle\rangle\big).$$

Since the agents of X would expect to have performed some action before or during a sequence of messages, their state may become inconsistent with an uncompromised trace. In fact, the only way the agents' states can become inconsistent with a trace set upon receiving an otherwise valid series of messages is if they causally precede those messages in that trace set. We formalize this in Lemma 14.

Example 3 in Section 2 makes use of causal precedence to observe misuse, as Alice would expect to have generated the nonce received in Bob's message if the adversary has compromised neither key. Note that in this case, Alice *does not* causally precede Bob's message if either of their keys has been compromised, so it is not possible for Alice to determine which key has been compromised if this occurs—that is, Alice's state will not be inconsistent with either trace set where one key is compromised, just inconsistent with traces where neither key is compromised.

Lemma 14 (Complete categorization)**.** *For a secret k, a set $X \subseteq \mathrm{Agent}$, and a trace set $T = Tr(P, \mathcal{A})$, let $tr \in T$ and $T_{uc} = \{t \mid t \in T \wedge t\!\upharpoonright_{c(k)} \in T\}$. If tr leaves the agents of X in a state inconsistent with any uncompromised trace, then compared to the trace set T_{uc}:*

i) a message in tr is not possible in any trace, or
ii) the message sequence observed in tr is contradictory, or
iii) X causally precedes the messages observed in tr.

Formally,

$$\neg\mathsf{consistent}(X, tr, T_{uc}) \implies \big(\exists m \,.\, \mathtt{recv}(m) \in tr \wedge \neg\mathsf{spec}(m, T_{uc})\big) \vee \mathsf{contra}(X, tr, T_{uc}) \vee \mathsf{causal}(X, tr, T_{uc}).$$

Proof. Assume this is not true, so that X is not causal, nor contains contradictory messages, nor are any of the messages impossible in an uncompromised trace. From this, we will reach a contradiction by constructing a trace in T_{uc} which leaves the agents of X in the same state as tr.

If the antecedent is false, then expanding the definitions,

$$\big(\forall \mathtt{recv}(m) \in tr \,.\, \mathsf{spec}(m, T_{uc})\big) \wedge \big(\exists tr' \in T_{uc} \,.\, tr|_X|_{\mathtt{recv}} = tr'|_X|_{\mathtt{recv}}\big) \wedge \big(\exists tr' \in T_{uc} \,.\, tr|_X|_{\mathtt{recv}} = tr'|_X|_{\mathtt{recv}} \wedge tr'|_X|_{\mathtt{send}} = \langle\rangle\big).$$

Thus, there exists a trace $tr' \in T_{uc}$ where $tr|_X|_{\mathtt{recv}} = tr'|_X|_{\mathtt{recv}}$ and $tr'|_X|_{send} = \langle\rangle$. Since this trace is in T_{uc} we can also conclude that $tr'\restriction_X \in T_{uc}$, as the only way the agents of X can influence the actions of the other agents or the adversary is through `send` events.

We can now concatenate all events of $tr|_X$ onto this trace. Each event is either local or relies on the state of the network; by the assumption above tr' must contain the same receive events, so either the network already contains or the adversary can already generate all of these messages after the trace $tr'\restriction_X$. Thus, $tr'\restriction_X \cdot tr|_X \in T_{uc}$, and given that all the state transitions of X are identical to those in tr, $\mathrm{state}(tr'\restriction_X \cdot tr|_X, X) = \mathrm{state}(tr, X)$. □

Note that in most cases, detection would only be feasible when the set of agents X that observes the misuse is a singleton. Nonetheless, knowing that some set of agents is able to observe misuse can be valuable for guiding protocol design, as it may be possible to modify the protocol so that these agents can communicate enough to detect, or to narrow the number of agents required to observe misuse. Alternatively, for some systems it may be practical to assume some out-of-band channel for communication between the observing agents, and perform detection that way.

As an example, if a protocol requires at least one agent from a set to make a request before a particular token is produced, then that set of agents collectively causally precedes the production of that token but none of the agents individually do. However, if the protocol can be modified such that the token produced depends on *which* agent requested it, then each agent individually could causally precede the production of their tokens. We distill lessons like these into general design principles and constructions below.

3.5 Design space

We now revisit our observations to identify the possible observation mechanisms and summarize design principles.

3.5.1 Main detection mechanisms

We identify three main mechanisms by which secret misuse can become observable. Ultimately, all of them rely on the observations that agents make through their interactions with the network. The difference in approaches mainly depends on the extent to which they take this information and their own actions into account.

Recall that state inconsistency is necessary but not sufficient for detection. Nonetheless, the categorization of observability conditions provides a categorization of the types of detection that can be designed into a detecting protocol.

1) **Trace-independent observability** of any single message in the trace set, which requires no knowledge of the prior trace events.
2) **Contradicting observations** when a sequence of observed messages can not occur in a single honest trace. This requires enough knowledge of prior observed messages to determine if new observations contradict.
3) **Acausal observations** when the observed messages contradict the agents' knowledge of their own activity. This is only possible for agents who causally precede the observed message sequence in honest traces, and it requires enough knowledge of past agent actions as well as prior observed messages to determine whether the agent caused the observed messages.

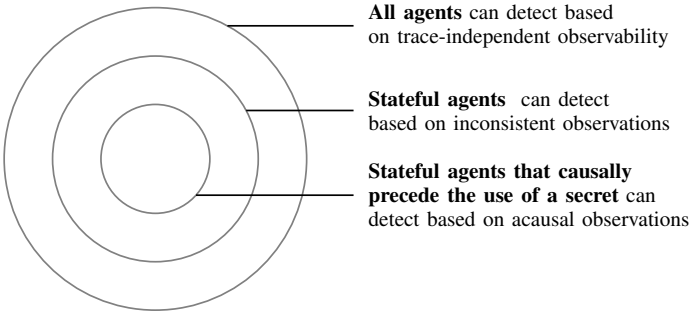

Figure 1. Venn Diagram of the type of agents and the detection mechanisms that they might be able to use. Only stateful agents that causally precede the use of a secret can use all three types.

3.5.2 Design principles

The combination of the three types of detection and the concrete detection mechanisms leads to a number of design principles for detecting the misuse of secrets.

We note that for any given application, there may be practical and security considerations that affect whether and how the principles can be applied. For example, the wish to maintain confidentiality and unlinkability of messages may limit the application of Principle 3. Restrictions on message size, communication complexity, and storage size may limit the applicability of any of the principles. This directly results in a trade-off between such restrictions and the ability to detect the misuse of secrets.

- **Principle 1**: Protocol messages should be tightly coupled to prior messages. This helps maximise the possibility of any misuse detection, and prevents an adversary from 'resychronizing' agents after misusing keys (e.g. the attack described in Example 2). Stateless protocols necessarily violate this principle.
- **Principle 2**: Include unique and unpredictable values in messages. This helps to establishing contradicting observations, and ensure an adversary can not correctly predict what an agent will do next. If values are not unique, then agents could get identical observations from messages sent at different points, making them indistinguishable. If an adversary could predict the next exchange, they could potentially carry it out in advance with one of the participants and then take their place in the real exchange without leaving any evidence.
- **Principle 3**: Maximize the spread of data that other parties might find contradictory or acausal. Detection requires observations, so it is important to increase the opportunities for that to happen. Ideally, some observations could be broadcast to all participants (e.g., used when disseminating transactions in Bitcoin-like systems [12,15,21] to detect double spending), but for many applications this is not feasible. This motivates the need for compromise solutions such as a gossip protocol (e.g., [9]).
- **Principle 4**: Identify which agents causally precede important messages, and ensure they can observe those messages. Agents who causally precede a sequence of messages can detect more than agents who can only detect by observing contradictions. It is therefore worthwhile to ensure that the protocol enables the detection of acausal observations as much as possible.
  For example, in the PKI setting, the agents with causal precedence are the domain owner and the CA, since a certificate for domain signed with a CA's key should only exist after it has been requested by the domain and then signed by that CA. If such a certificate occurs without the request, or without the CA signing it, then the key must have been misused. This principle is implicitly used in systems like Certificate Transparency [17] and other systems based on transparency overlays, which we will return to in Section 4.3.

Some minor aspects of the above principles are similar to principles from earlier work [1], but there are crucial differences. Principle 1 explicitly requires state, which leads to a trade-off between security guarantees and keeping track of state. Principle 2's unique values have been suggested before, but not all messages need to have unpredictable values for other security properties. This unpredictability is specifically useful for detection. Principle 3, which suggests spreading data, improves detectability at a clear cost in terms of transmissions, which would be avoided by previously proposed principles (except perhaps accountability). To the best of our knowledge, Principle 4 is entirely derived from our detection-based observations, though it is implicitly used in some systems.

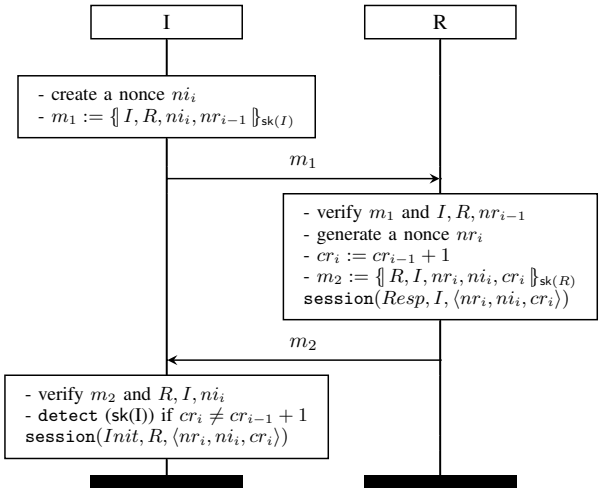

Figure 2. The counter protocol. The inclusion of nr_{i-1} in m_1 could instead be provided as a nonce in an additional message $m_0 = \{\!|\, R, I, nr_i \,|\!\}_{\mathsf{sk}(R)}$, as in the Keyless SSL example in Figure 3

4 Applications

The design principles discussed in Section 3.5 are general and can be used to improve existing systems in practice. In this section, we illustrate this by applying the techniques from Sections 3.4 and 3.5 to construct example protocols, and use them as guides to modify existing real-world protocols, including Keyless SSL [10], the Common Access Card (CAC)-based physical access control [23], and the certificate creation procedures of CAs. We show how these protocols can make use of misuse detection methods to be resilient against compromise. Finally, at the end of the section, we provide a collection of other popular systems, and briefly discuss each of them. In particular, we show how they fit into our design principles, and how our design principles can improve some of them.

4.1 Counting precedence and Keyless SSL

To illustrate the application of the principles, we imagine a simple protocol in which one agent increments a counter each time the other provides a fresh signature, similar to example 2. In this setting, the causal structure is very clear: the counter value is increased at most once for each signing key use.

The *counter protocol* is based on this idea, shown in Figure 2. Note that throughout this section, a message m signed by using signing key sk is presented as $\{\!|\, m \,|\!\}_{sk}$, which includes both the signature on m and the plaintext message m.

Despite its simplicity, the counter protocol has a number of desirable detection properties, which we formally verified with the TAMARIN prover [19], a tool for symbolic analysis of security protocols. In its framework, properties are expressed in a fragment of first-order logic that allows quantification over timepoints. We provide the full models in [2].

1) *Soundness if* $\mathsf{sk}(R)$ *cannot be compromised*: detection events of a term imply it was compromised, for all traces in $T = Tr(P, \mathcal{A})$ where $\mathcal{A}$ cannot compromise $\mathsf{sk}(R)$.

$$\forall t \in T, i, k \,.\, t_i = \mathtt{detect}(k) \implies (\exists j < i \,.\, t_j = \mathtt{compromise}(k)) \,.$$

   Note that if $\mathsf{sk}(R)$ can be compromised, $\mathtt{detect}(\mathsf{sk}(I))$ instead implies that $\mathsf{sk}(I)$ or $\mathsf{sk}(R)$ is compromised, which may still be a useful property.
2) *Detection guarantee for past sessions if* $\mathsf{sk}(R)$ *cannot be compromised*: In the trace set $T = Tr(P, \mathcal{A})$ where $\mathcal{A}$ cannot compromise $\mathsf{sk}(R)$, if there is a matching session in which I did not detect, then every session before matched.

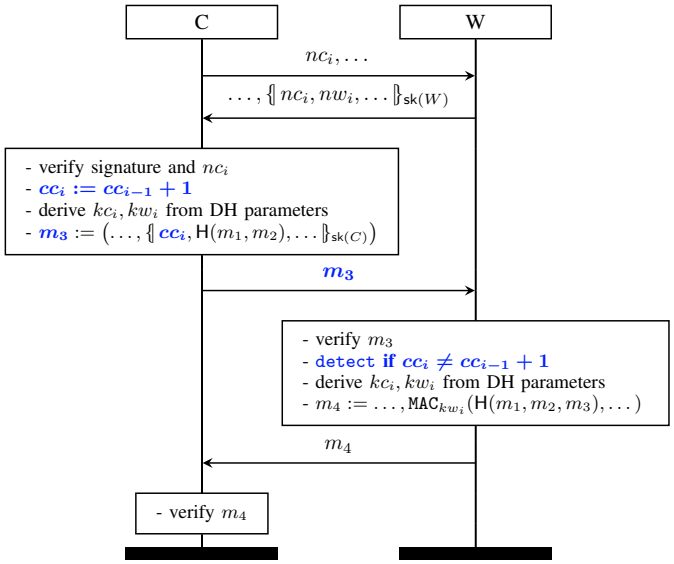

Figure 3. An example of the additions to the i-th session of the TLS mutually-authenticated key exchange in Keyless SSL. For clarity, we omit terms in the messages that are not relevant. Our modifications are highlighted in bold blue.

$$\begin{aligned}\forall t \in T, I, R, i_1 < i_2 < i_3, data_1, data_2 \,.\\ t_{i_1} = \mathtt{session}(Init, R, data_1) \wedge t_{i_2} = \mathtt{session}(Resp, I, data_2) \wedge\\ t_{i_3} = \mathtt{session}(Init, R, data_2) \wedge \neg(\exists k, i_1 < j < i_3 \,.\, t_j = \mathtt{detect}(k))\\ \implies (\exists j < i_1 \,.\, t_j = \mathtt{session}(Resp, I, data_1))\,.\end{aligned}$$

Keyless SSL. Keyless SSL was designed by CloudFlare to allow the provision of CDN services to web services that cannot or do not want to cede their certificates' private keys to CloudFlare [10]. In Keyless SSL, CloudFlare's servers interact with a customer-provided key server, which decrypts pre-master secrets as needed for CloudFlare to carry out key exchanges as if they knew the customer's private key.

In practice, this means that a large number of different private keys are each sufficient to use the customer's key server as an oracle, with much greater control over key issuance and revocation than in a typical TLS environment. This makes detection of key misuse especially valuable.

Our modified Keyless SSL protocol, shown in Figure 3, satisfies equivalent detection properties to the counter protocol, namely, both soundness and a detection guarantee when W is uncompromised. In [2] we provide symbolic verification, as well as an alternative protocol which allows C to detect instead of W. The assumption that W is uncompromised is reasonable considering W's role as a signing oracle. If W were compromised as well, then it is possible for the adversary to avoid detection by playing to role of W to resynchronize C. As discussed, this could be remedied by requiring W to provide a signed fresh token to be returned by C in the next session, however this requires an additional signing operation and provides benefit only when W might also be compromised.

Practical implications. The implementation of our modified protocol allows the CDN's customers to have assurance that either they have not been used to sign requests for an adversary that has gained access to a valid CDN server key, or if they were then the misuse of the key will be detected in short order. Furthermore, customers can immediately revoke to limit their risk, without requiring other parties to act. The proposed protocol requires very little modification and minimal storage requirements: a single counter value for each CDN server.

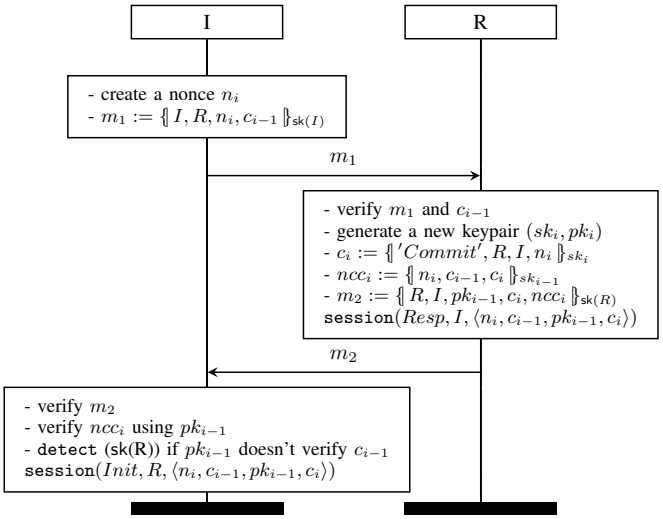

Figure 4. A commitment protocol.

4.2 Commitment and the Common Access Card

Principles 1 and 4 suggest that the message sequences in a protocol are tightly coupled. This can be achieved, e.g., by having each session contain a pre-commitment to some aspect of the next session, ideally with a commitment that can only be fulfilled with knowledge of the agent's state (to limit the risk of compromised state).

We show an example of such a construction in Figure 4, with an example of a *commitment protocol*, and an application in a high-security environment where the detection of cloning is valuable. R generates an asymmetric key pair and presents I with a fresh commitment constructed by signing session data with the secret key, as well as the secret key used for the previous commitment to ensure continuity. In the session following, R provides the public key that allows I to verify that the commitment is correct based on previous session data. R never reveals the commitment key, and hence the adversary can't authenticate their own session data even if they trick R into revealing an arbitrary number of commitments and proofs.

This commitment protocol has a number of desirable properties, which we also formally verified using TAMARIN. We include the model of this protocol in [2].

1) *Soundness*: A detection event implies compromise.

$$\forall t \in T, i, k \,.\, t_i = \mathtt{detect}(k) \implies (\exists j < i \,.\, t_j = \mathtt{compromise}(k)) \,.$$

2) *Detection guarantee against key compromise*: Against an adversary compromising both $\mathsf{sk}(I)$ and $\mathsf{sk}(R)$, when I completes a session with R and some data, then either R also completed with that data or I detected they did not.

$$\forall t \in T, I, R, i, data \,.\, t_i = \mathtt{session}(Init, R, data) \wedge \neg(\exists j < i \,.\, t_j = \mathtt{detect}(\mathsf{sk}(R))) \implies \exists j < i \,.\, t_j = \mathtt{session}(Resp, I, data).$$

3) *Detection guarantee after an uncompromised session*: Against an adversary who can reveal all agent state, if there was a previous correct session and the adversary has not revealed R's state since that session, then any session I completes with R will also be correct or I will detect.

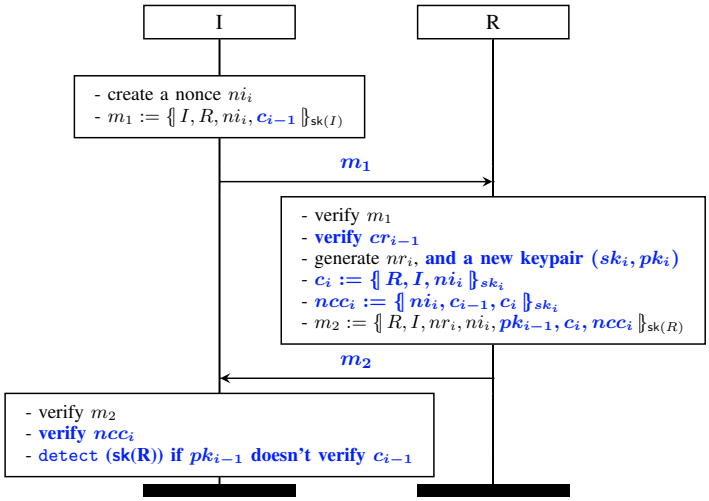

Figure 5. The i-th session of the modified ISO-IEC 9798-3-3 standard protocol. Modifications are highlighted in bold blue.

$$\begin{aligned}\forall t \in T, I, R, j_1 < i_1 < i_2, data_1, data_2\ .\\ t_{i_1} = \mathtt{session}(Init, R, data_1) \wedge t_{j_1} = \mathtt{session}(Resp, I, data_1) \wedge\\ t_{i_2} = \mathtt{session}(Init, R, data_2) \wedge \neg\big(\exists s, i_1 < c\ .\ t_c = \mathtt{compromise}(s)\big)\\ \implies (\exists j_1 < j_2 < i_2\ .\ t_{j_2} = \mathtt{session}(Resp, I, data_2)) \oplus\\ (\exists d < i_2\ .\ t_d = \mathtt{detect}(\mathsf{sk}(R))).\end{aligned}$$

Common Access Card. The Common Access Card (CAC) is the standard identification card for United States Defense personnel. The CAC has been used as an authentication token for security network systems and also for physical access to sensitive areas [7]. It supports asymmetric key cryptography and has writable memory. The CAC provides a useful example of a high-security domain where it is valuable both to detect if a cloned card has previously been used, as well as 'heal' compromise so that any clone becomes useless unless used immediately. To show an implementation, we exhibit a modified ISO-IEC 9798-3-3 protocol (Figure 5) in the scenario of the CAC physical access control. Here, the initiator I is a card reader which is connected to a back-end server, and the responder R is the CAC.

In this scenario, the modified protocol provides both detection of acausal action and of contradicting commitments, even if the attacker can extract all information from the CAC. In other words, the provided security guarantee is that when an attacker has a cloned copy of a CAC at time t, and used the cloned card at time t', then if the original card has been used in the time interval between t and t', the cloning of the card will be detected. If the original card is not used in the time interval between t and t', then the attacker can use the card to get access, but the cloning attack will be detected as soon as the original card is used again.

The modified ISO-IEC-9798-3-3 protocol achieves equivalent detection properties to the commitment protocol. These have been formally verified using TAMARIN, and the model of this protocol is included in [2].

Note that while this protocol requires three signing operations on the part of the card, two of these are by temporary commitment keys which only need to remain secure until the next authentication. As such, a weaker and faster signature computation could be used for these to reduce the computation required by the card.

Practical implications. We show with a modified standard protocol that it is possible for a smartcard authentication protocol to not only swiftly detect and revoke cloned cards, but also invalidate any existing clones every time a card authenticates. This is done in such a way that an attempt to use an earlier clone results in immediate detection and revocation of privilege *before* the card successfully

authenticates. Furthermore, this is possible even with the key of the reader compromised, *and* messages between the card and the reader intercepted.

4.3 Improving detection in transparency overlays

Transparency overlays and related public log-based systems [3, 4, 8, 17, 24, 25] are designed to make participants' behaviour public through the use of a third-party log, enabling misuse detection on the basis of acausal observations. To avoid having to trust the log maintainer, transparency overlays set up the log structure so that the maintainer must be able to prove that any two authenticated log states are consistent with each other. This allows a compromised log to be detected through observation of contradictory log states, and this misuse can be proven to other participants.

Participants examine log entries and detect if it contains entries they know it should not. For example, detection of a misissued certificate in Certificate Transparency may be done by domain owners checking the log and discovering a log entry for a certificate that they did not request. As discussed in Section 3.4, this cannot be done by any party that would not necessarily causally precede that certificate's issuance, nor can any misuse be proven to other parties. The detecting party can revoke the certificate as invalid, but there is no evidence that the CA's key has been misused to produce it. In practice, it is assumed that misuse of the CA's key would instead be determined manually based on multiple independent—or suspicious enough—certificates requiring revocation.

Though transparency overlays make use of contradicting observations to detect misuse of a log server's key, they rely entirely on acausal observations to detect misuse of a secret belonging to any of the parties committing to the log. From Section 3.4, we know that if a submitting party was compromised and their secrets were misused to authenticate (valid) submissions to the log, the only way for the participants to detect this misuse (without causal precedence) is through observing entries that are contradictory. Currently, however, applications like CT have no standard way that two otherwise valid entries in the log can contradict each other.

Based on our design Principles 1 and 2, we propose that CT-like transparency applications can be extended to allow dependencies between submissions from the same source, adding a further line of defense to transparency overlays and improving attribution when misuse is detected. Taking Certificate Transparency as a canonical example, we propose to add into each certificate a value dependent on previous certificate submissions. For example, the value could be the number of certificates n, indicating that this is the n-th certificate authenticated by that CA, starting at 0; or it could be the hash of certificate $n - 1$.

Contradiction testing. With an addition that allows log commitments to contradict each other, a log server can determine whether the CA knew about the previous commitment authenticated by them, or whether it might have been committed without their knowledge.

When contradictory certificates are submitted to a log server, the log server can swiftly notify the CA that either its key has been compromised or its system has not updated with issued certificates. On the other hand, if all certificates in the log are consistent with each other, then a domain owner discovering a misissued certificate for its domain in the log knows that the CA's own system must have been updating their state when the certificate was issued–an indication that the CA should have some record of issuing that certificate.

Implementation considerations. Our proposed additions (as applied to CT) make the CAs stateful in their creation of certificates, though with negligible overhead introduced. Importantly, the state kept by the CA depends only on local operations, and not on any feedback from log servers, so no latency is introduced into the process of issuing certificates. If all new certificate submissions to the log in Certificate Transparency were required to include this information, it would immediately benefit detection of CA key compromise.

This proposal is only one example of an addition which would force contradictions between log submissions from the same submitter in a transparency overlay. More elaborate constructions like the consistency proofs used by log servers could be leveraged to make submissions to a log from a misused key contradict a larger set of prior entries, for additional redundancy or for tying together multiple independent logs. In other transparency overlay applications, the commitment protocol shown in Section 4.2 could be used to ensure that future log submissions come from the same party that generated the pre-commitment in the prior entry.

4.4 Analysing other system designs

As mentioned and illustrated previously, our design principles are general and can be used to improve existing systems in practice. Here, we collect existing work that already conforms in part with the foundations and principles discussed in this paper. We show how existing systems fit into our design principles, and how they can be further improved by applying our work where relevant.

RFID tag cloning detection. Mechanisms [5, 18, 26, 27] for detecting cloned RFID tags in the supply chain have been widely studied. In [27], the RFID readers write random values to RFID tags as they pass through the supply chain so that the tag accumulates a sequence of random values. Cloned tags are then detected by observing contradicting sequences for the same tag identity.

This design follows both **Principle 1** and **Principle 2**. The tags are written with random values, and the sequence of values grows longer each time a reader is passed, making it very likely that a cloned tag will exit the supply chain with a different sequence written to it than the original.

More complex solutions could give stronger guarantees, but the resource constraints of RFID tags make it difficult to suggest further improvement.

The Double Ratchet Algorithm. The Double Ratchet algorithm [20] is designed for messaging systems to prevent replay, reordering or deletion of messages while encrypting with forward-secrecy in an asynchronous setting. Every message sent and received is encrypted with a new ephemeral symmetric key generated from two interlocking key ratchets, one of which is iterated with each message sent and the other when a message is received. A compromised message key will not help an attacker decrypt messages exchanged in previous sessions, and an adversary making use of a compromised message key causes the newly derived key to differ between the communicating agents.

The design of the double ratchet derives new keys each message, but this is still vulnerable to a persistent MITM attacker who was able to compromise both keys at some prior time. This could potentially be remedied by applying **Principle 3** (for example, through the use of the second concrete mechanism we describe). This would allow communicating agents to confirm that they agree on the keys being used, though at the cost of some privacy; care would have to be taken to anonymize log entries, etc.

Key-evolving cryptosystems. Key-evolving cryptosystems (e.g. [6, 13, 14]) were proposed to mitigate damage from compromised secret keys, through the use of periodic key refreshment. In the symmetric setting, a sender and a receiver share an initial long-term secret from which they derive a set of keys valid for a certain (application-specific) time period. In the asymmetric setting, one party holds only the public part of another party's private key, and updates it when they see the use of a new private key without further communication.

Though key-evolving cryptosystems have desirable properties, they could be improved through an application of our design principles. For example, by ensuring that key changes cannot be reset to any previous key (**Principle 1**) through some derivation process that relies on the prior keys.

TPM authentication protocol. The Trusted Platform Module (TPM) [22] is a chip designed to allow platforms to provide better security guarantees by securing cryptographic keys in its shielded memory. The authorisation protocols use 'rolling nonces' to prevent replay attacks: in each new session, the nonces generated in the previous session will be included in the authenticating MAC.

The use of unique nonces follows our design **Principle 2**, though an adversary who could inject messages would not be prevented from making use of the TPM and then injecting a message to resynchronize the nonces between the client and TPM. This could be prevented through the application of **Principle 1**, by deriving future nonces from past sessions so that an adversary cannot resynchronize them.

5 Related work

We present related work on security guarantees after key compromise, and on protocols with accountability and verifiability.

Post-compromise security. In [11], Cohn-Gordon *et al.* introduce *post-compromise security*: security guarantees for communication after a party's long-term keys are compromised. This is accomplished using dynamic secrets, similarly to the commitment protocol above (though the secrets in the commitment protocol are used only for authentication).

Post-compromise security as described differs from detection in what is done after attempting to establish a 'correct' session fails. If a guarantee of security is the only objective, then it makes sense to simply not allow a session that uses an incorrect key even if the long-term key is correct; doing so,

however, discards information that may be sufficient to determine the compromise of a long-term key. Detection and post-compromise security are therefore–while conceptually similar–orthogonal in nature and can be realized independently.

Accountability and verifiability. Küsters, Truderung, and Vogt have proposed definitions of accountability and verifiability [16] which aim to be widely applicable. The proposed definitions share some similar intuition with ours, i.e., they aim to discover if something went wrong. However, there is both a *conceptual difference* and a *technical difference* between their work and ours. The technical difference is that they use a process calculus as their underlying programming framework, which abstracts away from program state. This makes it very difficult to model the stateful protocols (such as the counter protocol) of our work. In particular, while they consider that an agent can run multiple programs, those programs do not share any state information. The conceptual difference is that they focus on misbehaving parties, while we focus on compromised parties. This appears not to be a huge difference, since a compromised party can readily be thought of as a special kind of misbehaving one that executes honest programs and dishonest ones. However, our detection protocols rely on the possibility of detecting that the attacker and the compromised party are not sharing the same state (e.g., the same counter). Because state is implicit in the framework of [16], this can't easily be captured.

For example, suppose a party is compromised by the attacker. Suppose also that the attacker wants to avoid detection, so runs only honest programs. In the [16] framework, the combination of the honest party and the attacker only runs honest programs, and hence cannot be detected. In reality, however, this attack could be detected, e.g., because of counter desynchronisation, and we capture this in our framework.

6 Conclusions

We have described and explored designs for protocols that detect when an adversary misuses an agent's secrets. Our design principles and constructions directly led to suggesting improvements for many deployed systems, enabling them to automatically detect the misuse of secrets. We have given example protocols and applications, described them systematically and verified their properties in the TAMARIN prover.

Concretely, our suggested improvements of existing systems such as CA's, the Common Access Card, or Cloudflare's Keyless SSL can significantly reduce the impact of a compromise, since they can be used to immediately revoke keys or shut down the related service.

There are some limitations to the proposed approaches. First, while our mechanisms are not applicable to *all* scenarios (e. g., because keeping synchronised state can be expensive or problematic in some use cases), it is clear that there are many applications whose security can be significantly improved by introducing these detection mechanisms. We therefore expect our mechanisms to find their way into many applications in the near future.

Acknowledgments. Benedikt Schmidt suggested the Keyless SSL case study as an application of our ideas in a discussion in 2014. We didn't anticipate it would become so topical.

References

[1] M. Abadi and R. M. Needham, "Prudent engineering practice for cryptographic protocols," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 6–15, 1996.

[2] Anonymous, "Full tamarin models of the submission," 2016, https://www.dropbox.com/s/hccnrf0xyn2a2l8/tamarin-submission.zip?dl=0.

[3] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: Attack resilient public-key infrastructure," in *Proc. of the ACM Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 382–393. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660298

[4] G. Belvin, "Key transparency overview," https://github.com/google/keytransparency/blob/master/docs/overview.md, 2016.

[5] E. Blass, K. Elkhiyaoui, and R. Molva, "Tracker: Security and privacy for RFID-based supply chains," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2011.

[6] K. Bowers, A. Juels, R. L. Rivest, and E. Shen, "Drifting keys: Impersonation detection for constrained devices," in *Proc. IEEE INFOCOM*, April 2013, pp. 1025–1033.

[7] M. R. Carr, "Smart card technology with case studies," in *Proc. International Carnahan Conference on Security Technology*. IEEE, 2002, pp. 158–159.

[8] M. Chase and S. Meiklejohn, "Transparency overlays and applications," in *CCS*, 2016, pp. 168–179.

[9] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of certificate logs," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, September 2015. [Online]. Available: http://www.netsec.ethz.ch/publications/papers/gossip2015.pdf

[10] CloudFlare, "Keyless SSL: The Nitty Gritty Technical Details," https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/, 2014.

[11] K. Cohn-Gordon, C. J. F. Cremers, and L. Garratt, "On post-compromise security," in *CSF, 2016*, 2016.

[12] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 45–59.

[13] J. Håstad, J. Jonsson, A. Juels, and M. Yung, "Funkspiel schemes: An alternative to conventional tamper resistance," in *Proc. of the ACM Conference on Computer and Communications Security*, ser. CCS '00. New York, NY, USA: ACM, 2000, pp. 125–133. [Online]. Available: http://doi.acm.org/10.1145/352600.352619

[14] G. Itkis, "Cryptographic tamper evidence," in *Proc. of the ACM Conference on Computer and Communication Security*. ACM Press, 2003, pp. 355–364.

[15] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 279–296.

[16] R. Küsters, T. Truderung, and A. Vogt, "Accountability: definition and relationship to verifiability," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, 2010, pp. 526–535.

[17] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," Internet Requests for Comments, RFC Editor, RFC 6962, June 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6962

[18] M. Lehtonen, F. Michahelles, and E. Fleisch, "How to detect cloned tags in a reliable way from incomplete RFID traces," in *RFID, 2009*. IEEE, 2009, pp. 257–264.

[19] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc.*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 696–701.

[20] Moxie Marlinspike and Trevor Perrin, "The double ratchet algorithm," https://whispersystems.org/docs/specifications/doubleratchet/, 2016.

[21] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.

[22] Trusted Computing Group, "TPM 1.2 Specification," Jul. 2014, https://www.trustedcomputinggroup.org/tpm-1-2-protection-profile/.

[23] United States Department of Defense, "DoD Common Access Card," May 2016, http://www.cac.mil/common-access-card/.

[24] J. Yu, V. Cheval, and M. Ryan, "DTKI: A new formalized PKI with verifiable trusted parties," *Comput. J.*, vol. 59, no. 11, pp. 1695–1713, 2016.

[25] J. Yu, M. Ryan, and C. Cremers, "Decim: Detecting endpoint compromise in messaging," *IACR Cryptology ePrint Archive*, vol. 2015, p. 486, 2015.

[26] D. Zanetti, L. Fellmann, and S. Capkun, "Privacy-preserving clone detection for RFID-enabled supply chains," in *RFID, 2010*. IEEE, 2010, pp. 37–44.

[27] D. Zanetti, S. Capkun, and A. Juels, "Tailing RFID tags for clone detection," in *20th Annual Network and Distributed System Security Symposium, NDSS*, 2013.