An extended abstract of this paper appears in the proceedings of IEEE CSF 2017. This is the full version.

Automatically Detecting Compromised Secrets: Foundations, Design Principles, and Applications

Version 2.0, 21 January 2026

(Version 1.0 was released in 2017. We give an overview of major changes in Appendix A.)

Kevin Milner[1], Cas Cremers[2], Jiangshan Yu[3], and Mark Ryan[4]

[1]University of Oxford
[2]CISPA Helmholtz Center for Information Security
[3]University of Luxembourg, Luxembourg
[4]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 compromised. Such constructions can be used, e.g., to automatically shut down compromised services, or to automatically revoke compromised 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 compromise 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 compromised secrets. 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 compromise 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 compromised keys. However, while these mechanisms typically provide a means to observe usage of keys, 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 was compromised 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 compromised 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 compromise 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 whether a Certificate Authority's key was compromised, 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 compromise 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 [12] to enable the customer to detect compromise 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 [33], and the certificate creation procedures of CAs.

We proceed as follows. In Section 2, we provide an informal introduction to the idea of compromise detection. We then, in Section 3, develop foundations for the automatic detection of compromised secrets and categorize the ways in which detection can occur. 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. We give an overview of major changes between the different versions of this document in Appendix A.

2 Foundations: an informal introduction

In order to investigate how we could detect compromise to allow automatic response, we first consider some motivating examples. 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; classical information is cloneable and so there is no information-theoretic consequence of an attacker learning it. 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 must be 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 those compromised secrets is when the attacker deviates—or rather, is forced to deviate—from the original owner's behaviour or ongoing actions, either because the dishonest behaviour is inherently and noticeably different or the ongoing actions create discord.

As we are interested in protocols which allow for automatic response, participants must be able to logically conclude that some deviation *must* be the result of a compromise, in order to ensure there are no false positives. This is notably different from the field of *anomaly detection*, which also seeks to identify deviation from "honest" actions, but does so probabilistically. The challenge in anomaly detection is generally in identifying actions which are *allowed* but *very unlikely* to be part of normal activity. Unfortunately this often leads to false positives,

and as such typically requires human oversight or only minor responses (e.g., requiring a user to re-enter their password).

Consider the following examples of protocols and attacks 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 3.5.

Example 1. *Alice has a secret* $\mathsf{sk}(A)$ *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* $\mathsf{sk}(A)$ *and* $\mathsf{sk}(B)$ *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 on the i-th session, she increments her counter, generates a new nonce* na_i *and includes them both in her message along with the last nonce received from Bob (*nb_{i-1}*). 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* nb_i*, his counter value* cb_i*, and Alice's last nonce* na_i*. The next message from Alice contains a new nonce* na_{i+1}*, nb, and an incremented counter value* ca_{i+1}*, the next message from Bob a nonce* nb_{i+1}*,* na_{i+1}*, and his counter value* cb_{i+1}*, etc. We show this protocol in Figure 1*

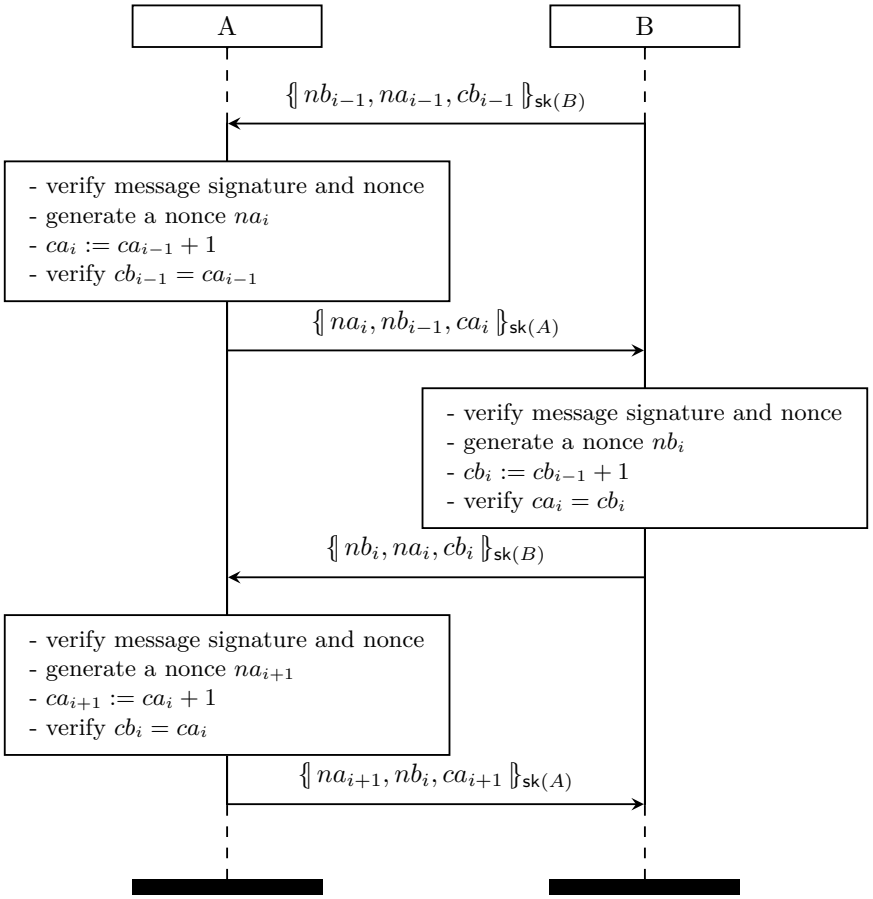

Figure 1: The protocol described in Example 2, beginning from the i-th session.

Example 2 illustrates a simple case in which compromise can be detected. If an attacker gains knowledge of $\mathsf{sk}(A)$ 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-synchronized and they could detect upon comparing these values that $\mathsf{sk}(A)$ was compromised. 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

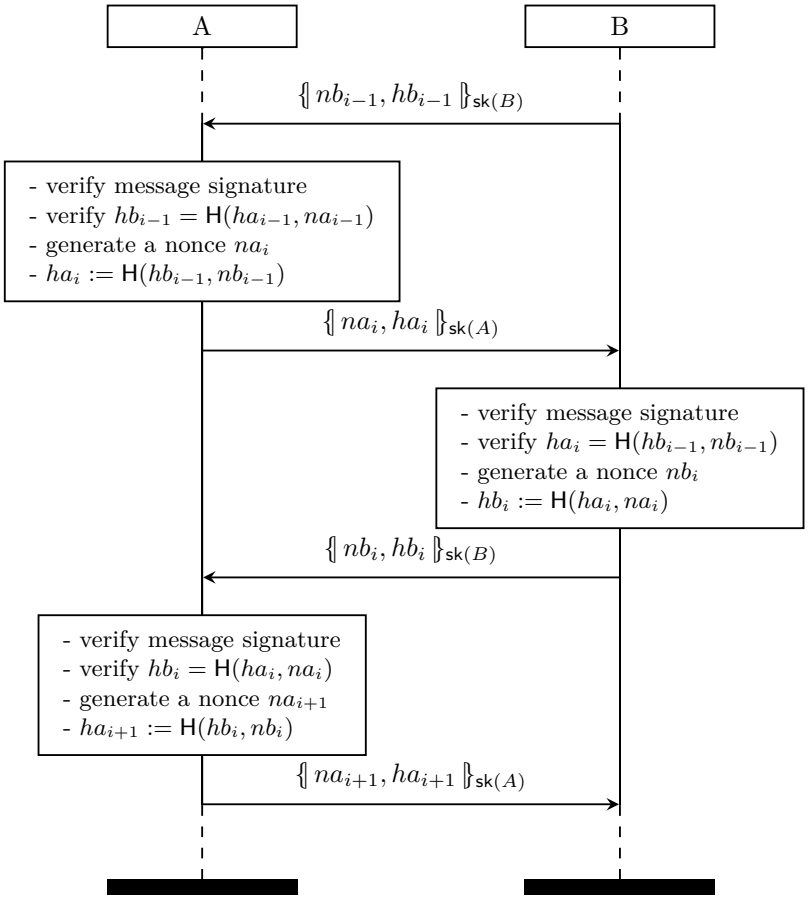

Figure 2: The protocol described in Example 3, beginning from the i-th session, assuming a function $\mathsf{H}(x, y)$ which extends a hash chain x with an additional term y.

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 in the i-th session, she includes a new nonce na_i 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 nb_i and extending the hash chain with na_i. The next message from Alice contains a new nonce na_{i+1} and the hash chain extended with nb_i, etc. We show the i-th session of this protocol in Figure 2.*

In Example 3, suppose an attacker obtains Alice's key $\mathsf{sk}(A)$ 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 $\mathsf{sk}(A)$ and $\mathsf{sk}(B)$ 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.

We will come back to these examples explicitly later on, but they motivate some intuition for how a coupling of information between past and present sessions allows adversary action to be noticed, at least in principle.

3 Foundations and design space

In this section, we develop formal foundations and explore the design space for detecting compromised secrets. While our contributions can be informally understood and applied in practice by skipping most of this section and immediately moving to Section 3.6, 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 3.4, 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 compromise can be observed into three broad categories in Section 3.5 and show that they together form a complete categorization. Finally, we combine these elements for the design space in Section 3.6.

3.1 Notation

Sequences are used throughout this section. Given a sequence s, we address the i-th element as s_i and use $idx(s) = \{1, \ldots, |s|\}$ to refer to the set of indices of s, where $|s|$ is the length of s. We use angle brackets for sequences, where $\langle\rangle$ denotes the empty sequence and $\langle e_1, e_2, \ldots, e_{|s|}\rangle$ is used for the sequence comprising element e_1 followed by e_2 and so on. The concatenation of two sequences s and s' is written $s \cdot s'$. We overload set notation for sequences and write $e \in s$ for a sequence s if and only if $\exists i \,.\, s_i = e$.

In order to discuss a particular subsequence, we use the sequence projection operator. For a sequence l and a set S, the projection $l|_S$ is defined as

$$l|_S = \begin{cases} \langle\rangle & \text{if } l = \langle\rangle \\ \langle l_2, \ldots, l_{|l|}\rangle|_S & \text{if } l_1 \notin S \\ \langle l_1\rangle \cdot (\langle l_2, \ldots, l_{|l|}\rangle|_S) & \text{if } l_1 \in S. \end{cases}$$

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.

3.2 Reasoning about protocols

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 detect compromise, and what is necessary for detection.

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 accessing an external resource—e.g. sampling the random number generator, or accessing the network to send or receive messages—or internally modifying their state, etc. From this, a transition system arises in which an agent with a particular state performs a specific action dictated by the protocol, which then results in a new agent state depending on the action and potentially the state of the network or the result of sampling the random number generator. We write Protocol to denote the set of all protocols.

More formally, the protocol dictates an action taken as a deterministic function of agent state. The possible states resulting from that action may rely on the state of the network, and may (if sending a message to the network) influence the state of the network. The result cannot depend on the state of other agents or the adversary directly, only the network, and we assume that agents can access the network only through `send` actions, which add a particular message to the network, or `recv` actions, which can receive a copy of any message from the network. In a particular execution, one of these possible states is chosen non-deterministically, and the combination of the action performed, the agent performing it, and the result is called an *event*. For example, we use $\texttt{recv}_x(m)$ to denote an event in which an agent x performed a `recv` action and received message m (though we will often drop the agent identifier or message if they are not relevant). Each agent x has a transition relation

$$\text{Step}_x \subseteq \mathit{State} \times \text{Event} \times \mathit{State},$$

where an agent event fully determines the resulting state transition, i.e.

$$(st, e, st') \in \text{Step}_x \wedge (st, e, st'') \in \text{Step}_x \implies st' = st''.$$

We will use these agent events, along with adversary events, as a record of protocol execution.

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 define events for the adversary similarly to agents above, and denote the set of all adversary models Adv.

The definitions above do 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 allows for the adversary to learn arbitrary terms (and thus all terms held in an agent's state). 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 events 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 $P \in \text{Protocol}$ and adversary model $\mathcal{A} \in \text{Adv}$ gives rise to a transition system with agent states, the network as a set of messages, and the state of the adversary. At each step, either an agent or the adversary performs an event, with a corresponding state transition, and possibly a change of network state. We log each adversary or agent event in a sequence called a *trace*. The particular representation of events in the trace is not important for our purposes; instead, we require only that the trace contains sufficient information to reconstruct the state of the adversary and every agent at each point in the trace based solely on the prior events ascribed to them in the trace, and the state of the network from all prior events. Thus, for a set of agents X with states $State_X = \{State_1, \ldots, State_{|X|}\}$, there is a transition relation

$$\text{Step}_{(P,\mathcal{A})} \subseteq (State_X, Network, State_{\mathcal{A}}) \times \text{Event} \times (State'_X, Network', State'_{\mathcal{A}}),$$

where transitions caused by events are restricted such that

1. if the event is performed by an agent x, then $(st, e, st') \in \text{Step}_x$ where st is the state of x in $State_X$. The new agent state $State'_X$ is obtained by replacing the state of A in $State_X$ with st', and $State'_{\mathcal{A}} = State_{\mathcal{A}}$,
2. if the event is $\texttt{recv}_x(m)$ for some agent x and message m, then $m \in Network$,
3. if the event is $\texttt{send}_x(m)$ for some agent x and message m, then $Network' = Network \cup \{m\}$ (note this is the only case in which an agent changes the state of the network),
4. if the event is performed by the adversary, then $(State_{\mathcal{A}}, e, State'_{\mathcal{A}}) \in \text{Step}_{\mathcal{A}}$, and $State'_X = State_X$.

We also require that events create deterministic state transitions, i.e.

$$(st, e, st') \in \text{Step}_{(P,\mathcal{A})} \wedge (st, e, st'') \in \text{Step}_{(P,\mathcal{A})} \implies st' = st''. \tag{1}$$

The execution of a protocol is an alternating sequence of state tuples and events,

$$\langle st_0, e_1, st_1, \ldots, e_n, st_n \rangle$$

where st_0 is the initial state and each tuple $(st_{k-1}, e_k, st_k) \in \text{Step}_{(P,\mathcal{A})}$. The corresponding trace is the sequence of events $\langle e_1, \ldots, e_n \rangle$. We call the set of all possible traces arising from some protocol and adversary model a *trace set*, and use $Tr(P, \mathcal{A})$ to refer to the trace set of a particular protocol P and adversary $\mathcal{A}$. Note that trace sets are prefix-closed, as individual transitions are assumed to be atomic and the participants can stop at any time.

Generally we do not care about the specific events 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. In addition to the special `send` and `recv` events which interact with the network, we name two other special types of events. We use $\texttt{compromise}(k)$ to refer to any adversary event that compromised some data k from any agent's state. This allows us to refer to, for example, the subset of traces in a trace set in which a particular term is never compromised. 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 both to the adversary and the agent (e.g. some settings may assume a public key infrastructure). The adversary's 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 [26]) defines how terms may be derived.

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 protocol* $P \in \text{Protocol}$ *soundly detects compromise with respect to an adversary model* $\mathcal{A} \in \text{Adv}$ *if*

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

Note that completeness, in the sense of always detecting after compromise, is not possible in general unless compromise is directly observable by protocol participants. Nevertheless, it is possible to give guarantees of detection under particular conditions: for example, in Section 4 we will discuss protocols that guarantee detection in the session following particular adversary activity.

For shorthand, we enumerate some of the common sequence projections that we will use throughout this section to isolate particular parts of traces:

- 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,
- $\restriction_{c(k)}$ for all events that are not $\texttt{compromise}(k)$ events,
- $|_{\texttt{send}}$ or $|_{\texttt{recv}}$ for all send or receive trace events respectively, and
- $|_{\texttt{net}}$ for all network trace events (i.e. including both send or receive trace events).

In this work we do not prescribe any specific 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.

3.3 Reasoning about agents

In order to reason about agent capabilities, we must be able to talk about their state as well as the possible events 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)**.** *For a set of agents $X \subseteq \text{Agent}$, we introduce the notation $\mathit{state}(tr, X)$ to represent the collective state of the agents of X after a trace tr.*

Proposition 6 (State convergence)**.** *Let T be a trace set and X a set of agents. Let $tr, tr' \in T$ be two traces such that $\mathit{state}(tr, X) = \mathit{state}(tr', X)$. Then*

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

Recall from Equation 1, events fully determine each individual state transition. Thus, by definition the state of some particular agents X after a trace tr can be reconstructed entirely from the events in $tr|_X$. Therefore, it is necessarily true that $\text{state}(tr, X) = \text{state}(tr', X)$ 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 $T \subseteq Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$. A protocol extension performed by a set $X \subseteq \text{Agent}$, beginning from a 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,*

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

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, we will be using these protocol extensions to represent what a set of agents could determine by running a protocol amongst themselves *after* a particular trace, in an ideal environment where no adversary interferes with them. This captures all the things the agents might collectively compute from their current state. State convergence can be leveraged to show a useful property of the protocol extensions across all possible protocols.

Lemma 8. *Let T be a prefix-closed set of traces such that $T \subseteq Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$, and let $X \subseteq$ Agent be a set of agents. Then*

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

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

Proof. Assume otherwise; that is, without loss of generality there is a sequence 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 $s|_X \neq s$, or $tr' \cdot s \notin T$, or there are `recv` events with no corresponding `send` in s. The first and last of these are trivially false by the requirement that $s \in SP(tr, T, X)$. Thus, it must be 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 e is a valid state transition after the state reached in tr but not after the state reached in tr', then from the definition of $\text{Step}_{(P,\mathcal{A})}$ it must be because either e is not a valid transition for the agent's state after tr' or e is an event $\texttt{recv}(m)$ for a message m that is not in the network after tr'. But both $\text{state}(tr, X) = \text{state}(tr', X)$, and the antecedent requires that all messages sent to the network are identical; in other words, the event e can be performed after tr', and thus $tr' \cdot \langle e \rangle \in T$.

By Proposition 6, $\text{state}(tr \cdot \langle e \rangle, X) = \text{state}(tr' \cdot \langle e \rangle, X)$. Since every trace is finite, the sequence s' must be shorter than s; so we recurse this argument over s'. If $s' = \langle\rangle$ then we are done, the trace suffix $\langle e \rangle$ was in both $SP(tr, T, X)$ and $SP(tr', T, X)$, a contradiction. If s' is not empty, then s' is a trace suffix in the set $SP(tr \cdot \langle e \rangle, T, X)$ but not $SP(tr' \cdot \langle e \rangle, T, X)$; we can repeat the argument above to remove the next event from the suffix and we find that every event of s is valid after tr'. □

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.

Lemma 9 (Equivalent definition of soundly detecting)**.** *For a detection protocol $P \in$ Protocol and adversary model $\mathcal{A} \in$ Adv,*

$$\begin{aligned}\mathsf{sound}(P, \mathcal{A}) \iff \forall tr, s \,.\, tr \in Tr(P, \mathcal{A}) \wedge s \in SP(tr, Tr(P, \mathcal{A})) \implies \\ \forall k \,.\, (\texttt{detect}(k) \in s \implies \texttt{compromise}(k) \in tr)\,.\end{aligned}$$

Proof. Recall the definition of sound,

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

If a protocol is sound by the definition above, then for all traces in $Tr(P, \mathcal{A})$, all $\texttt{detect}(k)$ events must be preceded by a $\texttt{compromise}(k)$ event. Since $tr \cdot s \in Tr(P, \mathcal{A})$ by the definition of a protocol extension, $\texttt{detect}(k) \in s \implies \texttt{compromise}(k) \in tr \cdot s$. Since $\texttt{compromise}(k)$ cannot occur in the protocol extension s by definition, it must be that $\texttt{compromise}(k) \in tr$.

In the other direction, let us assume that the implication is false: that `detect` events in protocol extensions imply a `compromise` event in the trace, but the protocol is not sound. If the protocol is not sound then, by the prefix-closed nature of trace sets, there must exist a trace $tr' \cdot \langle \texttt{detect}(k) \rangle \in Tr(P, \mathcal{A})$ ending in a `detect` event, where $\texttt{compromise}(k) \notin tr'$. Now consider the protocol extensions $SP(tr', Tr(P, \mathcal{A}))$. By definition, the event $\texttt{detect}(k)$ must have been performed by one of the agents, and it is not a `recv` event; further, by our assumption, $tr' \cdot \langle \texttt{detect}(k) \rangle \in Tr(P, \mathcal{A})$. But then $\langle \texttt{detect}(k) \rangle$ must be a valid protocol extension of tr', and so tr' must contain $\texttt{compromise}(k)$—a contradiction. □

3.4 Observing usage of a compromised secret

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 subsequently attribute it to the compromise of a secret, the protocol (or in a wider sense, the security mechanism) must be designed to make usage of the compromised secret observable by the detecting 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 usage of a compromised secret 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 compromised secrets if that compromise had no effect on them. We formalize this below, using the protocol extension properties discussed above.

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

$$\forall s, tr' \,.\, s \in SP(tr, T, X) \wedge \texttt{detect}(k) \in s \wedge tr' \in T \wedge \left(tr'|_{c(k)} = \langle\rangle\right) \wedge state(tr, X) = state(tr', X) \implies \neg\mathsf{sound}(P, \mathcal{A}).$$

That is, if a set X of agents detect the misuse of a compromised secret k in a trace $tr \in Tr(P, \mathcal{A})$ when their state could also be reached in a trace tr' without compromise, then the protocol cannot be sound. This means that sound detection is necessarily impossible in a completely stateless protocol where $\forall tr \,.\, state(tr, X) = state(\langle\rangle, X)$, as well as after any situation in which the adversary can 'reset' an agent's state back to a state reachable in an uncompromised trace.

Proof. Assume it is possible for the agents in X to soundly detect after tr, and thus there exists a suffix $s \in SP(tr, T, X)$ where $\texttt{detect}(k) \in s$.

The antecedent requires a trace tr' where

$$tr' \in T \wedge \left(tr'|_{c(k)} = \langle\rangle\right) \wedge (\text{state}(tr, X) = \text{state}(tr', X)),$$

and from Lemma 8,

$$SP(tr, T, X) \subseteq SP(tr', T, X);$$

thus $s \in SP(tr', T, X)$. 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)**.** *Let $T \subseteq Tr(P, \mathcal{A})$ for some protocol P and adversary $\mathcal{A}$. The state of some agents $X \subseteq$ Agent after a trace tr is consistent with the trace set T if there is at least one trace in T which leaves the agents in X in the same state as tr.*

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

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

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

Note that the set $\{t \mid t \in T \wedge t\!\upharpoonright_{c(k)} \in T\}$ includes traces involving compromise of the key, so long as the compromise was not necessary for any events in the trace; this is to differentiate mere *compromise* (without usage, which cannot be detected) from *misuse* (which might be detectable). Unobservable key compromise necessarily 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 compromise. Doing so would also detect in the trace $tr\!\upharpoonright_{c(k)}$, since by definition

$$\exists tr' \,.\, \text{state}(tr, X) = \text{state}(tr', X) = \text{state}(tr'\!\upharpoonright_{c(k)}, X).$$

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.

3.5 Categorizing observable use of compromised secrets

Lemma 10 shows that a set $X \subseteq$ 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 compromised secrets.

We divide the ways of observing the use of compromised secrets 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 to receive, could not be received 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.

3.5.1 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 event category 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 whether a message would occur in any trace within an arbitrary trace set T.

Definition 12 (Allowed messages)**.** *A message m is allowed in a trace set T if there exists a trace in T containing a receive event of that message. Formally,*

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

Messages which are not allowed in a trace set are referred to as *disallowed* messages. For example, recall Example 1 from Section 2 in which a key is used to generate a message that would never occur when the parties involved follow the protocol.

This type of observability is relatively trivial. In practice one can rarely rely on the adversary sending a message which is, in itself, evidence of a compromised secret. Nevertheless, it represents a special case of observable inconsistency with a trace set, in which no knowledge except the message itself is required to determine inconsistency.

3.5.2 Observing contradictions

If the messages in a trace are contradictory compared to a trace set that sequence of messages cannot occur in any trace of the trace set. This is formalized with the predicate contra.

Definition 13 (Contradictory messages)**.** *Given a set $X \subseteq$ Agent, a trace tr, and a trace set T, we say that the agents of X have received a contradictory sequence of messages when*

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

Note that receiving any disallowed message in a trace implies that the message sequence in that trace is contradictory, i.e. for any set $X \subseteq$ Agent, trace tr, and trace set T,

$$\big(\exists m \,.\, m \in (tr|_X)|_{\mathtt{recv}} \wedge \neg \mathsf{spec}(m, T)\big) \implies \mathsf{contra}(X, tr, T).$$

Recall Example 2 in Section 2, in which two agents exchange messages with increasing counter values. An adversary taking the place of an agent temporarily can be detected in this case because it is possible to observe contradictory messages, as each message received from the other agent is expected to include an incremented counter value. An agent receiving two messages with the same counter value could observe that they contradict, even if each message would be allowed in the trace set where neither agent is compromised.

A stronger example making use of contradictory messages to detect is found in transparency overlays like Certificate Transparency, which we will discuss further in Section 4.1. Briefly, the 'log' in a transparency overlay signs particular messages for the protocol participants which are expected to be mutually consistent; compromise of the log's key could therefore be detected by receiving two such signed messages that contradict each other.

3.5.3 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 compromise of their key if they receive a message which uses it that is not in their state. This extends in more complex ways: the transparency overlays we will discuss in Section 4.1 are based on the ability of an identity or domain owner to determine when an entry in the log exists without some action on their part, on the assumption that only the owner should be initiating the process which leads to an entry in the log.

We define a notion of *violating causality*, where an agent can observe that the messages of a trace violate causality if they have some guarantee that they are required to participate in particular ways every time some sequence of messages occurs in the trace set.

Definition 14 (Violation of causality)**.** *The messages of a trace tr in a trace set T are said to* violate causality *for a set of agents X if there is some trace in T in which those messages can be received, but no trace in T in which the same sequence of sent and received message occurs. Formally,*

$$\mathsf{acausal}_{(}X, tr, T) \equiv \big(\exists tr' \in T \;.\; (tr|_X)|_{\mathtt{recv}} = (tr'|_X)|_{\mathtt{recv}}\big) \wedge \big(\forall tr' \in T \;.\; (tr|_X)|_{\mathtt{net}} \neq (tr'|_X)|_{\mathtt{net}}\big).$$

Since the agents of X would expect to have performed some particular actions 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 observe a violation of causality for that trace set. We formalize this in Lemma 15.

Example 3 in Section 2 can observe the use of a compromised secret because of the causal connection between messages. In the trace set where the adversary has not compromised either agent's key, the sequence of messages Alice receives from Bob are dependent on her sent messages, and thus a mismatched hash chain value would be evidence that their trace is inconsistent with that trace set.

Note, however, that in both the trace set where the adversary may compromise Alice's key, and the trace set where the adversary may compromise Bob's key, Alice cannot determine if Bob's hash chain value violates causality. In the former, the adversary can sign a different nonce value to Bob with Alice's key, and in the latter the adversary can sign a message as Bob directly with a different hash chain value. Thus, it is not possible for Alice to differentiate between these two trace sets upon receiving the wrong hash chain value: the trace is consistent with both.

Lemma 15 (Complete categorization)**.** *For a secret k, a set $X \subseteq$ Agent, and a trace set $T = Tr(P, \mathcal{A})$, let $tr \in T$ and $T_{uc} = \{t \mid t \in T \wedge t\!\uparrow_{c(k)} \in T\}$. If tr leaves the agents of X in a state inconsistent with any uncompromised trace, then compared to the trace set T_{uc}:*

i) the message sequence observed in tr is contradictory, or

ii) the message sequence observed in tr violates causality.

Formally,

$$\neg\mathsf{consistent}(X, tr, T_{uc}) \implies \mathsf{contra}(X, tr, T_{uc}) \vee \mathsf{acausal}_{(}X, tr, T_{uc}).$$

Proof. Assume this is not true, so that agent state after the trace is inconsistent with any uncompromised trace, but the trace does not contain contradictory messages nor does it violate causality. 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.

Note that T is generated by a protocol, so it is prefix-closed and thus by definition $\langle\rangle \in T_{uc}$. As such, tr must be non-empty, or it would be consistent with T_{uc}.

If the consequent is false, then expanding the definitions,

$$\big(\exists tr' \in T_{uc} \;.\; (tr|_X)|_{\mathtt{recv}} = (tr'|_X)|_{\mathtt{recv}}\big) \wedge \Big(\big(\forall tr' \in T_{uc} \;.\; (tr|_X)|_{\mathtt{recv}} \neq (tr'|_X)|_{\mathtt{recv}}\big) \vee \big(\exists tr' \in T_{uc} \;.\; (tr|_X)|_{\mathtt{net}} = (tr'|_X)|_{\mathtt{net}}\big)\Big).$$

Thus, there exists some trace $tr' \in T_{uc}$ such that

$$\left((tr|_X)|_{\mathtt{net}} = (tr'|_X)|_{\mathtt{net}}\right).$$

If agent state after the trace tr differs from all traces in T_{uc}, then since trace events are by definition sufficient to construct the state of all agents and the adversary, there must be at least one event in tr performed by the agents of X which differs from their events in all traces of T_{uc}. Thus, there is some non-empty prefix $p \cdot \langle e\rangle$ of $tr|_X$ such that

$$(\exists tr' \in T_{uc} \;.\; tr'|_X = p) \wedge \neg\left(\exists tr' \in T_{uc} \;.\; tr'|_X = (p \cdot \langle e\rangle)\right),$$

where p may be the empty sequence and e is a single trace event. We separate the possible events for e into events that are not `recv` events, and those that are, and show that both lead to a contradiction.

Take any trace $tr' \in T_{uc}$ such that $tr'|_X = p$, which must exist from the above. If e is an event performed by an agent in X, but is not a `recv` event, then (by the assumption that agents can only interact through the network) it depends only on the local state of the agent. Since $\text{state}(p, X) = \text{state}(tr', X)$,

$$\exists tr \in T \;.\; tr|_X = (p \cdot \langle e\rangle) \implies (tr' \cdot \langle e\rangle) \in T.$$

But since $tr' \in T'$, and e cannot be a compromise event, then by definition

$$(tr' \cdot \langle e\rangle) \in T \implies (tr' \cdot \langle e\rangle) \in T_{uc},$$

a contradiction.

If e is instead a `recv` event performed by an agent in X, then it depends on both the state of the agents as well as the state of the network adversary. Since $p \cdot \langle e \rangle$ is a prefix of $tr|_X$, and there exists some trace in the prefix-closed set T_{uc} with all the same network events of tr by our original assumption, there must be some $tr_{uc} \in T_{uc}$ such that

$$((p \cdot \langle e \rangle)|_X)\,|_{\texttt{net}} = (tr_{uc}|_X)|_{\texttt{net}}.$$

In other words, it must be possible for the adversary to construct the message received in e in tr_{uc} without compromise.

Now, take any trace $tr' \in T_{uc}$ such that $tr'|_X = p$. Note that $(p|_X)|_{\texttt{net}} = (tr'|_X)|_{\texttt{net}}$ by definition. Since agents can only influence the adversary's state through `send` events, and those are identical for the agents of X in p and tr', the only way the receive event e might not be valid after tr' is through the activity of the other agents not in X.

Using this, we construct our contradiction. Consider a trace tr'_{uc} constructed from tr_{uc} by replacing the events by the agents in X with those in $tr'|_X$ in order, with the constraint that the `send` and `recv` events performed by X occur in the same places as before. This trace must exist in T_{uc}, as all events local to X are still valid regardless of the other agent activity, and all network events of X remain identical. But then, by construction, $tr'_{uc}|_X = (p \cdot \langle e \rangle)|_X$, a contradiction. □

Note that in most cases, detection would only be feasible when the set of agents X that observes the use of a compromised secret is a singleton. Nonetheless, knowing that some set of agents is able to observe usage of a compromised secret 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 compromise. 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 have a causal role in the production of that token (and could therefore detect based on violations of that causality) 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 have a causal role in the production of their own tokens. We will now distill lessons like these into general design principles and constructions.

3.6 Design space

We now revisit our results on categories of compromise observation to identify detection mechanisms and summarize them into some design principles for detection protocols. Finally, in Section 4.1 we re-examine transparency overlays with the additional context our foundations provide, and identify potential improvements.

3.6.1 Main detection mechanisms

In Section 3.5 we categorized three main mechanisms by which the compromise of a secret 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 implies a categorization of the types of detection that can be designed into a detecting protocol, and some necessary conditions.

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 cannot 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 are in a position to observe violations of causality compared to 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.

3.6.2 Design principles

The combination of the three types of detection leads to a number of design principles for detecting compromised secrets.

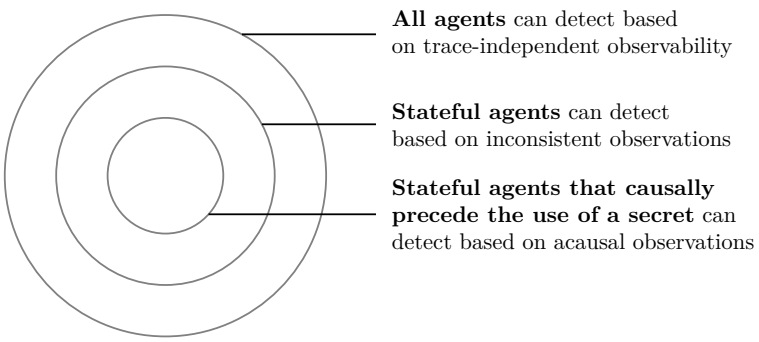

Figure 3: Venn Diagram of the type of agents and the detection mechanisms that they might be able to use. Only stateful agents that have a causal role the use of a secret could make use of all three types.

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 compromised secrets.

- **Principle 1**: Protocol messages should be tightly coupled to prior messages. This helps maximize the possibility of any compromise 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 cannot 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 [17, 22, 30] 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., [10]).
- **Principle 4**: Identify which agents can observe violation of causality for important messages, and ensure they can observe those messages. Agents who are required to trigger 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 which can observe violations of causality are the domain owner and the CA, since a certificate for a 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 compromised. This principle is implicitly used in systems like Certificate Transparency [24] and other systems based on transparency overlays, which we will return to in Section 4.1.

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.

Next, we will apply the intuition and principles built in this Section to reconsider a class of existing designs for detection, called transparency overlays, and develop novel protocols to detect compromised secrets.

4 Applications

With the foundations established above, we now have the intuition necessary to analyze existing protocol designs and instantiate new protocols with detection properties. In this section, we will begin in Section 4.1 by examining transparency overlays, and propose a concrete improvement based on our work. We then present two types of concrete protocols which can detect compromise: one based on counters, and the other based on so-called commitments. We consider these example protocols as a template for modifying existing protocols, and demonstrate this for each protocol by examining a scenario where compromise of secrets is a concern and modifying an existing protocol to achieve detection properties.

In Section 4.2 we present a type of detecting protocol based on counters. We present an example protocol which has minimal state and communication requirements, while still providing a detection guarantee when an adversary is only capable of compromising a subset of keys. This is often the case in real systems, and we take a recent development in Content Delivery Networks (CDNs), a protocol called Keyless SSL [12], as a case study. We discuss Keyless SSL in more detail in Section 4.2.2, but in short Keyless SSL is designed to allow a customer of a CDN provider to maintain control over the private keys for their web certificates, by having the customer provide a signing oracle for the key. We show that by embedding our counter-based example inside the key exchange used by Keyless SSL, it is possible for the customer to detect some compromise of, and automatically revoke access by, a CDN server's key, with minimal overhead.

Finally, we describe a type of detecting protocol based on providing commitments to the contents of other sessions. We give an example protocol which provides detection guarantees even when both parties involved may be compromised, and can detect an adversary attempting to authenticate as an agent unless they have compromised the full state of that agent since their last session. We consider this example in the context of systems that require high assurance, and in particular the Common Access Card (CAC) [33], the standard identification for United States Armed Forces personnel. Our commitment-based detection protocol can detect and revoke a cloned card on the next use of the original, while also preventing the stockpiling of cloned cards for use at a later time. We demonstrate this by modifying a standard authentication protocol used by smart cards, ISO-IEC 9798-3-3 [20].

We formally verified several of our case studies with the Tamarin prover [26], 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 [28].

For our Tamarin models, we consider an arbitrary (unbounded) number of agents and sessions. We only restrict the models in the sense that each agent executes sequentially. More precisely, an agent doesn't run two sessions concurrently with the same peer. In all other respects our models are as accurate as possible within the symbolic setting.

To help Tamarin prove the properties, we manually formulated several invariants (referred to as reusable lemmas in the Tamarin framework). Once these are formulated, Tamarin automatically proves the invariants and uses them to prove the desired properties, i.e., no interaction is required. For more information about the formal verification of these case studies, see [27]

4.1 Improving transparency overlays

Transparency overlays and related public log-based systems [2, 4, 9, 24, 36, 37] are designed to make participants' behaviour public through the use of a third-party log, enabling compromise detection on the basis of acausal observations. To avoid having to trust the log maintainer, transparency overlays make use of a log structure where the maintainer must be able to prove that any two log states they authenticate are consistent with each other. This allows misbehaviour by the log to be detected through observation of contradictory log states, and furthermore this compromise can be proven to other participants. In fact, this is precisely our **Principle 3** – to distribute information as widely as possible – and is one of the core developments underlying transparency overlays.

With an authenticated log that allows compromise of the log's key to be detected, it becomes possible to build a system in which acausal actions by another party can be observed. Participants making use of a transparency overlay can examine log entries to ensure both that every relevant action they see has an entry in the log, and that all entries in the log appear correct. This allows participants to observe acausal entries in the log, even where they would normally not be a part of a session making use of it. For example, detection of a mis-issued 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 mentioned in **Principle 4** above.

Transparency overlays thus have two interlinking parts which make use of detection: the first comparing the ongoing authenticated log states to ensure they do not contradict each other (which can be done by anyone), and the second auditing the contents of the log for entries indicating acausal use of a secret (which can be done only by particular parties for each entry). In Certificate Transparency, these are typically performed by individual users' browsers while navigating the web, and by domain owners respectively, as shown in Figure 4.

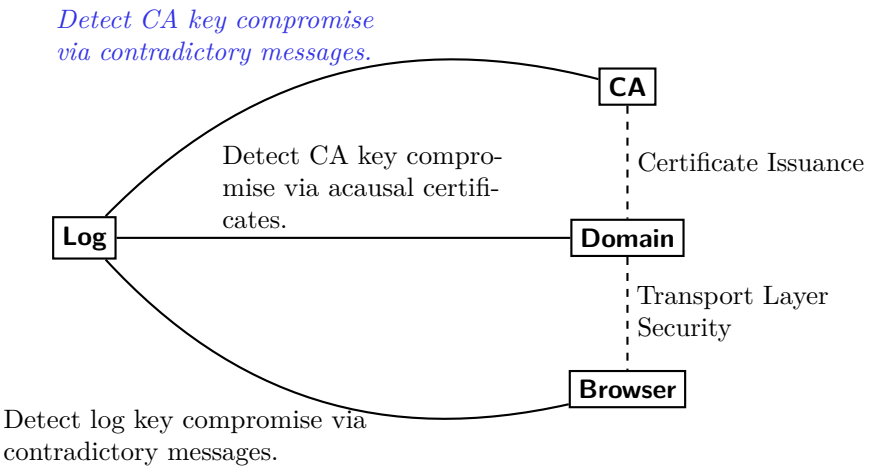

Figure 4: Certificate Transparency's interactions with existing web PKI (shown dotted lines), with the types of detection employed. We mark our proposed modification in *blue italics*.

Recall from our foundations above that observing contradictory messages requires less than observing acausality. Since we can detect acausal uses of the CA's key, what about contradictory uses? Surely if we can perform the former then there must be some modification which allows for the latter (though it might be impractical to implement).

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 compromise is detected. Taking Certificate Transparency as a canonical example, we propose to add into each log submission a value dependent on the previously submitted certificate (for example, the hash of the previous submission) so that the log can perform a first-line check for compromise. We show this modification highlighted in Figure 4.

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. If the certificates do not contradict each other, then this is evidence that the certificate authority's state has been updated with each certificate issued. On the other hand, if they do contradict each other, this indicates that either the CA's key has been compromised by an adversary, or the CA systems are failing to correctly track each certificate they issue.

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 been updated with all issued certificates. On the other hand, if all certificates in the log are consistent with each other, then a domain owner discovering a mis-issued 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; arguably, CAs are already expected to keep an internal audit trail for each certificate issued.

Importantly, the state kept by the CA depends only on local operations, and not on any feedback from log servers. No latency is introduced into the process of issuing certificates, a facet that is vital for the modification to be practical. 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 compromised 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 4.3 could be used to ensure that future log submissions come from the same party that generated the pre-commitment in the prior entry.

In practice, implementing this change in systems using transparency overlays would mean that compromise of a key to submit to a log could be detected rapidly if there are still ongoing honest submissions. Furthermore, it would give some assurance that all log entries were at some point known to the submitter, since they must have updated some part of their state with each previous submission. Though this modification does not replace the need for detection of acausal activity in the log, the benefits it provides can be done with negligible overhead on the part of both the log submitters and the log itself. The change would enhance the ability of transparency

overlays to provide detection guarantees, and narrow down the potential issues to investigate if compromise is detected.

4.2 Counter-based detection

A simple approach to ensuring a causal relationship between messages in an authentication protocol is to have parties count the number of successful authentications. If an agent has some causal role in counter increases when the key is uncompromised, then when they observe violations of causality they can determine that a key was compromised.

This simple approach of counting authentications has been applied in other domains in the past. An early example can be found in the late 17th century, with a lock designed by the locksmith John Wilkes [35] to count the number of times it has been opened. This allows an owner who remembers the value of the counter each time they open it to be sure that only they have opened the lock. More recently, a similar construction of a physical lock was patented in 1974 [7], likewise with the goal of allowing the owner to detect when other parties successfully 'authenticated' to the lock.

Here, we present a simple counter-based detection protocol based on asymmetric operations, and discuss verification of the protocol's properties in Tamarin. There are several scenarios where this protocol can serve as a blueprint for modifying existing protocols to provide detection. As a case study, we take an example relevant to internet infrastructure: the Keyless SSL protocol introduced by CloudFlare [12]. We describe the Keyless SSL protocol in Section 4.2.2 and present a modified handshake that provides detection guarantees.

4.2.1 A counter-based detection protocol

The *counter-based* detection protocol shown in Figure 5 is based on the notion of causality violation discussed in Section 3.5. The intuition behind the protocol is to ensure that counter increments are caused by recent uses of a signing key. The initiating agent I increments their counter only when they receive a freshly-signed message (i.e., including a nonce generated by I) from the responder R, and R only updates their stored counter value when they receive a similarly freshly-signed message from I to complete the session. With this, it becomes possible for I to observe acausal messages, either due to R's counter incrementing without I having sent a corresponding signed message, or due to I updating the stored counter value without a corresponding increment signed by R.

The counter-based protocol shown also provides a useful illustration of how our foundations clarify the limitations of a protocol. In the trace set where neither $\mathsf{sk}(I)$ or $\mathsf{sk}(R)$ are compromised, the responder R can determine if either of the messages m_1 or m_2 violate causality, which includes the incremented counter value to be checked, and the initiator I can do so for m_2 before completing the session (and thus before beginning the next session with an incremented counter). Because of this, it is possible for I to soundly determine that a key must have been compromised if they observe a mismatch between their stored counter, and a counter increment signed by R.

However, m_2 containing an incremented counter without action by I does *not* violate causality in the trace set where $\mathsf{sk}(I)$ may be compromised (even if $\mathsf{sk}(R)$ cannot be), because the adversary can take on I's role in the protocol. Further, it would not violate causality in the trace set where $\mathsf{sk}(R)$ may be compromised (even if $\mathsf{sk}(I)$ cannot be) as the adversary can simply generate m_2 on their own without action by I. Therefore, it is impossible for I to determine *which* of the two keys were compromised from observing an acausal counter value.

From our foundations, it also becomes clear how the protocol could be modified in order to allow this: I must be required to send messages for all responses from R in m_2, even when $\mathsf{sk}(R)$ is compromised. This could be done, for example, by having R also include the signed message it received from I in the prior session. An adversary misusing $\mathsf{sk}(I)$ in this modified protocol would lead to the real I being presented with an incorrect counter value along with a message signed by $\mathsf{sk}(I)$ that I never sent. Misusing $\mathsf{sk}(R)$, on the other hand, would lead to an invalid counter update combined with a signed message that was sent by I.

We present here the simpler version of the protocol, as the proposal above still fails to detect when both keys may be compromised. In many practical situations, such as the Keyless SSL case we examine later, only one key is of specific concern, and in Section 4.3.1 we present a protocol which can detect compromise even when both keys are compromised.

The counter protocol has the following detection properties.

Sound detection If there is a `detect` event triggered for a particular key pairing, then at least one of the keys in that pairing was compromised before the `detect` event. Formally, this is stated as

$$\forall k_I, k_R, t_1 \,.\, \texttt{detect}(\mathsf{pk}(k_I), \mathsf{pk}(k_R)) \;@\; t_1 \implies (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \texttt{compromise}(k_I) \;@\; t_0) \vee (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \texttt{compromise}(k_R) \;@\; t_0)\,.$$

Note that this also implies soundness in traces where compromise is restricted to one of the keys.

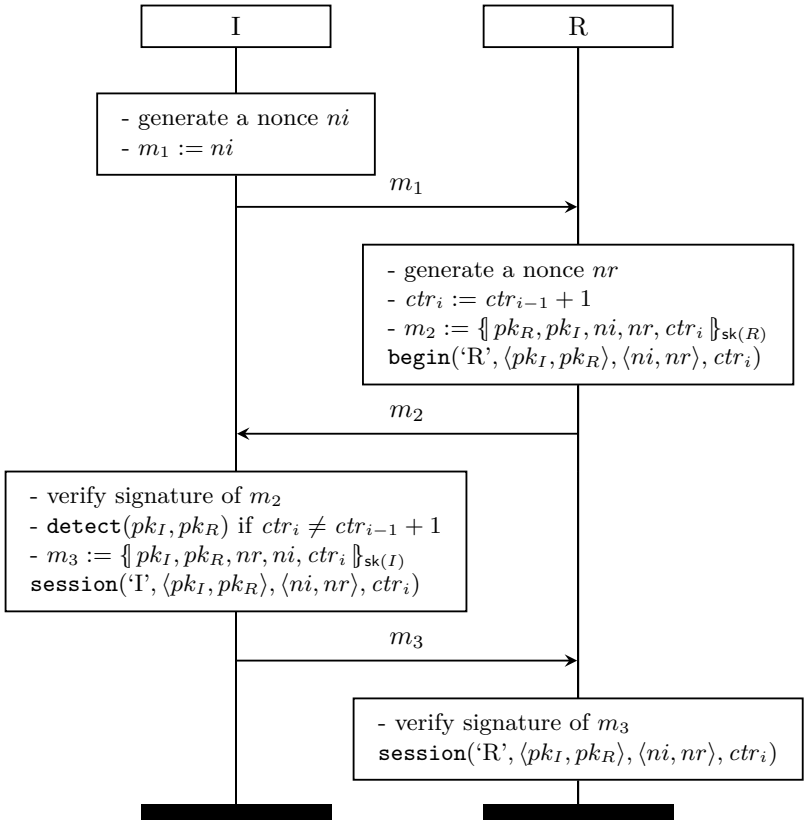

Figure 5: The i-th session of the counter-based detection protocol. The initial message with ni can be removed at the cost of additional state, if the initiator instead generates the nonce for message m_2, and the responder begins with m_1 by including the ni from the previous session.

Agreement without compromise If no keys were compromised, then a session event by R implies a matching session event by I, with agreement on nonces and counter value. This is just a basic authentication property, ensuring that the protocol is correct in the uncompromised setting. Since counter values are unique, this also implies injective agreement.

$$\forall k_I, k_R, data, t_1 \,.\; \mathtt{session}(\text{'R'}, \langle \mathsf{pk}(k_I), \mathsf{pk}(k_R)\rangle, data) \mathbin{@} t_1 \wedge \neg\,(\exists k, t_c \,.\; \mathtt{compromise}(k) \mathbin{@} t_c) \implies \exists t_0 \,.\; t_0 \lessdot t_1 \wedge \mathtt{session}(\text{'I'}, \langle \mathsf{pk}(k_I), \mathsf{pk}(k_R)\rangle, data) \mathbin{@} t_0.$$

Guaranteed detection of compromise of the initiator's key in prior sessions Assuming k_R cannot be compromised, if I completes a session at t_3, then either every prior session completed by R has a corresponding session completed by I, or I will detect compromise of k_I. Formally,

$$\forall pk_I, pk_R, data_1, data_2, t_1, t_2, t_3 \,.\; t_1 \lessdot t_2 \wedge \mathtt{begin}(\text{'R'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_2 \wedge \mathtt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_3 \wedge \mathtt{session}(\text{'R'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_1 \wedge \neg\big(\exists t_c, k_R \,.\; pk_R = \mathsf{pk}(k_R) \wedge \mathtt{compromise}(k_R) \mathbin{@} t_c\big) \implies \big(\exists t_0 \,.\; \mathtt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_0\big) \vee \big(\mathtt{detect}(pk_I, pk_R) \mathbin{@} t_3\big).$$

Together, these properties show that it is possible to instantiate useful detection storing only a counter value at each agent. This counter-based protocol provides a basis for modifying existing protocols achieving injective agreement to also detect compromised secrets, with minimal overhead. We now examine one such case, that of Keyless SSL.

4.2.2 Keyless SSL

Content delivery networks (CDNs) are services which provide many geographically dispersed caching proxies for internet content. CDNs are designed to increase availability and performance of internet services, by providing redundancy of content hosting and lower latency to end-users (because of geographic proximity), and they can do so mostly transparently, even as a third-party [16]. This has made CDNs extremely popular, either as a service that can be purchased from CDN providers or as company-specific infrastructure for many internet companies; CDNs carried over half of all internet traffic in 2016 [11].

Purchasing CDN services from a third party, however, presents an obstacle to the use of TLS. Since the CDN's server comes between the end-user and the origin of any requested content, the CDN must be able to terminate TLS connections as if it were the origin. Naïvely, this would mean that each of the CDN's servers must have a private key corresponding to a certificate for the origin domain—a worrying proposal for some of their customers, and potentially even illegal in some jurisdictions and sectors. To alleviate these concerns, Keyless SSL was developed.

Keyless SSL is a protocol designed by a CDN provider, CloudFlare, to allow the provision of Content Delivery Network (CDN) services to their customers which have domains that do not want to, or cannot cede the private keys associated with their certificates to CloudFlare [12]. In Keyless SSL, CloudFlare's servers interact with a key server provided by their customer (e.g. a web service), using it as an oracle to complete a key exchange on that web service's behalf. This allows CloudFlare to carry out TLS handshakes as if they knew the web service's private key, while it remains secure on the web service's hardware.

In practice, this means that a large number of different private keys are each sufficient to use that web service's key server as an oracle, though with much greater control over key issuance and revocation than in a typical TLS environment. This makes detection of key compromise especially valuable: key revocation and remediation are much easier than they would be for a compromised domain certificate. By modifying the Keyless SSL protocol to include the same concepts as the counter-based example above, we can ensure that events from the web service must send a message to cause counter increments, so that they can detect if the CDN server's private key has been compromised in any prior session. Since a CDN server carries out a new TLS handshake with the web service's server frequently (roughly every two hours), any compromise of the private key can be swiftly detected, and the key can be revoked by the web service before completing the session.

As in the example counter-based protocol above, the detection guarantee in our modified protocol requires that only one participant is compromised. Specifically, we assume that the oracle's key is uncompromised; this is justified by their role as a signing oracle in the Keyless SSL protocol. If both the CDN and the web service can be compromised, then it is possible for the adversary to avoid detection by re-synchronizing the CDN server's counter, running sessions as the web service. But in this case, the adversary could also use the web service's key directly to impersonate them without involving the CDN.

The implementation of the modified protocol presented here would allow the web services hosted by the CDN to have assurance that their key server has either not been accessed by an adversary who has gained access to any of the CDN server keys, or if it is being accessed by the adversary then it will be detected in short order. Furthermore, the web service can revoke a detected key immediately and automatically before completing the key exchange, preventing further damage from the compromised key. The proposed protocol requires very little modification and minimal storage requirements: a single counter value for each CDN server.

We briefly describe the protocol flow shown in Figure 6 before discussing its properties below. We consider the mutually authenticated TLS handshake performing a Diffie-Hellman exchange (DHE) between a CDN server C (the initiator), and one of their customers, a web service owner W[1]. C and W hold secret keys $\mathsf{sk}(C)$ and $\mathsf{sk}(W)$, respectively. They also have some means to validate each other's public keys–typically, $\mathsf{pk}(W)$ would be provided through some authenticated out-of-band communication while $\mathsf{pk}(C)$ is signed by a CDN-specific CA known to W. In this setting, we wish to provide some security guarantee against an attacker who obtains $\mathsf{sk}(C)$ and all state information (i.e. nonces) of C generated in any session. The goal of our protocol modification is to detect the compromise of $\mathsf{sk}(C)$.

In the first session, the counter begins at some known value, say '0'. In the i-th session, when C is establishing a shared secret with W, C begins a mutually-authenticated TLS exchange by creating a new nonce nc_i and sending the first message (known as the `ClientHello` message in the TLS protocol) to W. Upon receiving this message, W generates its own nonce nw and replies to C with, among many other things, a signature on nc and nw in the second message. Note that the exchange so far is unmodified from the standard TLS mutually-authenticated DHE.

Upon verifying the signature in the second message, C is certain that $\mathsf{sk}(W)$ is being actively used in the current session, and so increments its counter ctr_i. This counter value is then included in m_3. This is the only modified message of the protocol.

[1]Keyless SSL allows only two cipher suites in the TLS handshake with the oracle, both of which use an elliptic curve Diffie-Hellman exchange [12], so we do not need to consider modifications or analysis of other modes.

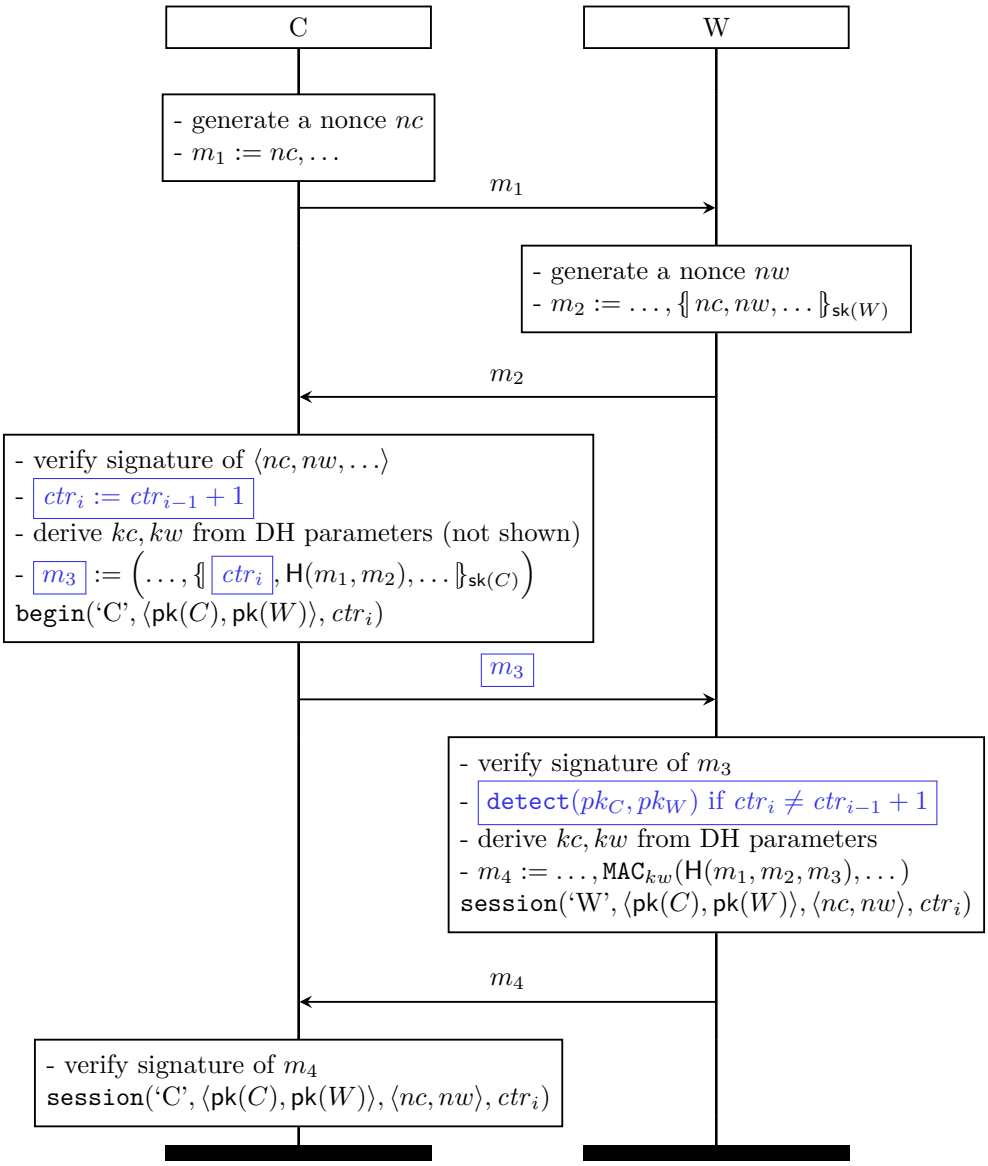

Figure 6: An example of the additions to the i-th session of the TLS mutually-authenticated key exchange used in Keyless SSL. For clarity, we omit terms in the messages that are not relevant. The modifications for compromise detection are boxed and highlighted in blue.

If ctr_i does not match what W expects but the hash and signature are valid, a detection event will be raised and W can revoke C's key immediately to limit potential damage. W can later contact C through an out-of-band channel to begin remediation and attempt to discover the source of the compromise.

The modified Keyless SSL protocol exhibits similar properties to the counter example above.

Sound detection If there is a `detect` event triggered for a particular key pairing, then at least one of the keys in that pairing was compromised before the `detect` event. Formally,

$$\forall k_C, k_W, t_1 \,.\, \mathtt{detect}(\mathsf{pk}(k_C), \mathsf{pk}(k_W)) \mathbin{@} t_1 \implies (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{compromise}(k_C) \mathbin{@} t_c) \vee (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{compromise}(k_W) \mathbin{@} t_0)\,.$$

Note that this also implies soundness in traces where compromise is restricted to one of the keys.

Agreement without compromise If no keys were compromised, then a session event by C implies a matching session event by W, with agreement on nonces and counter value. This ensures that the protocol is correct in the uncompromised setting. Since counter values are unique, this also implies injective agreement.

$$\forall keys, data, t_1 \,.\, \texttt{session}(\text{`C'}, keys, data) \ @\ t_1 \wedge \neg\,(\exists k, t_c \,.\, \texttt{compromise}(k) \ @\ t_c) \implies \\ \exists t_0 \,.\, t_0 \lessdot t_1 \wedge \texttt{session}(\text{`W'}, keys, data) \ @\ t_0.$$

Guaranteed compromise detection of the CDN key in prior sessions Assuming the web service's key k_W is not compromised, if W completes a session at time t_3 then either every prior session completed by W has a matching session completed by C, or W will detect compromise of k_C at t_3. Formally,

$$\forall pk_C, pk_W, data_1, data_2, t_0, t_2, t_3 \,.\, t_0 \lessdot t_3 \wedge \texttt{begin}(\text{`C'}, \langle pk_C, pk_W\rangle, data_2) \ @\ t_2 \wedge \\ \texttt{session}(\text{`W'}, \langle pk_C, pk_W\rangle, data_2) \ @\ t_3 \wedge \texttt{session}(\text{`W'}, \langle pk_C, pk_W\rangle, data_1) \ @\ t_0 \wedge \\ \neg\,(\exists t_c, k_W \,.\, pk_W = \mathsf{pk}(k_W) \wedge \texttt{compromise}(k_W) \ @\ t_c) \implies \\ (\exists t_0 \,.\, \texttt{session}(\text{`C'}, \langle pk_C, pk_W\rangle, data_1) \ @\ t_1) \vee (\texttt{detect}(pk_C, pk_W) \ @\ t_3).$$

4.3 Commitment-based detection

A more complex approach to a detection protocol is one applying ideas from key rotation schemes in order to facilitate detection as well as prevent adversary action. In this section we present a protocol based off an agent generating a new key each session, and providing a 'commitment' to that secret. In the following session, the agent must provide proof that they still have knowledge of the secret in order to authenticate. If an adversary has not recently compromised the agent, then they will not know the secret necessary to fully authenticate; if they have, then the next honest session will be using the previous secret, allowing for compromise detection.

The technique presented in the commitment-based protocol below is ideal for situations where a high degree of security is required. We consider one particular scenario as a case study, the use case of the Common Access Card. The Common Access Card (CAC) is the standard identification card for United States Defense personnel, and has been used as an authentication token for security network systems and also for physical access to sensitive areas [8]. 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 immediately used.

As the protocols used by the CAC are not public, we show the applicability of our commitment-based protocol to this domain by modifying and verifying a protocol used in similar cards, the ISO-IEC 9798-3-3 authentication protocol [20]. In this instance, the initiator I in the 9798-3-3 protocol is a card reader connected to a back-end server, and the responder R is the CAC.

4.3.1 A commitment-based protocol

The design implications in Section 3 suggest that the message sequences in a protocol should be tightly coupled. Taking this to an extreme, in the commitment-based protocol every session includes not only a term established in the previous session, but also a term which commits the agent to some aspect of the *next* session. Naïvely, this might be done by generating a random value and providing the output of a one-way function in one session, followed by presenting the original value in the following session. But while this may be sufficient when there is no adversary on the network, in a Dolev-Yao setting the adversary could make use of the revealed value to insert their own session. Instead, we prevent this with a construction which allows an agent to commit to some property of the next session for which the corresponding proof is both coupled to the session data, and does not reveal the means to construct a different proof.

We show an example of such a construction with the commitment-based detection protocol shown in Figure 7. In this protocol, 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. Note that at the time the commitment is made, I is not capable of validating it directly, beyond knowing that it's associated with the previous commitment through the signature under the proof key. In the session following, R provides the public key that allows I to verify that the commitment is correct based on previous session data.

At no point does R reveal the key used to construct the commitment, ensuring that the adversary cannot authenticate their own session data even if they trick R into revealing an arbitrary number of commitments and their proofs. Instead, the proof is provided for the previous commitment while 'liveness' of the corresponding secret is proven in the next session, by using it to sign the next commitment.

To analyse this protocol, we augment the adversary's capability with a rule which outputs the commitment signing key (the only non-public portion of I's state) to the adversary, labelled as `compromiseState`. Our final

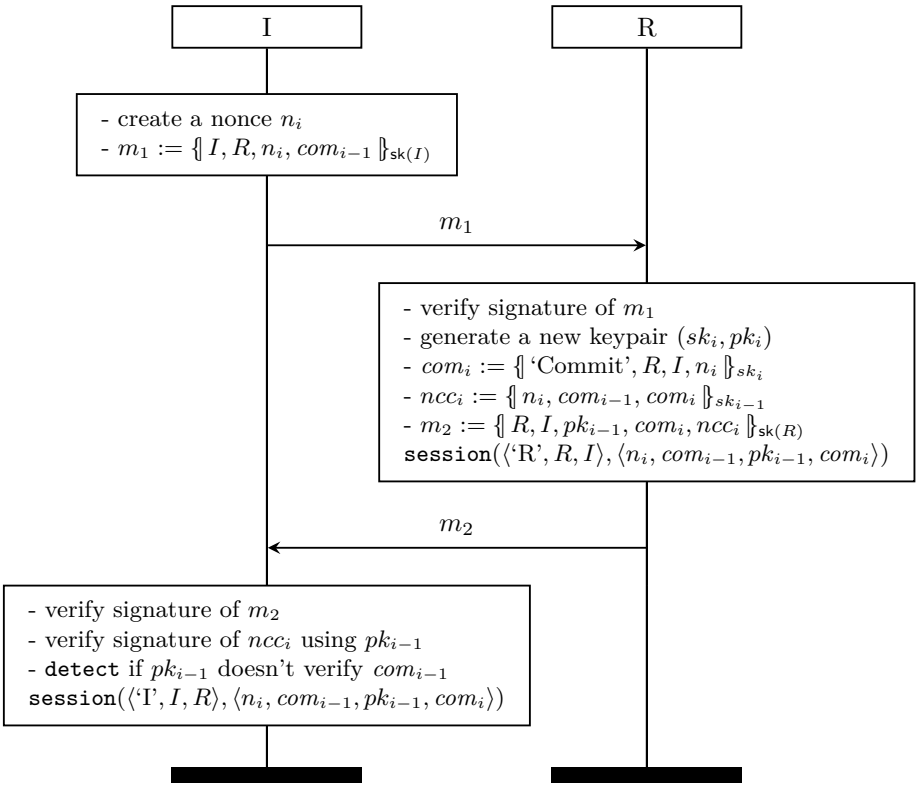

Figure 7: The i-th session of our example commitment-based detection protocol.

security property is expressed with respect to when this action occurs. Further detail of our analysis can be found in [27]. The commitment protocol has a number of desirable detection properties.

Sound detection If there is a `detect` event triggered for a particular key pairing, then the responder's key was compromised before the `detect` event. Formally,

$$\forall k_I, k_R, t_1 \,.\, \mathtt{detect}(\mathsf{pk}(k_I), \mathsf{pk}(k_R)) \ @\ t_1 \implies (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{compromise}(k_R) \ @\ t_0)\,.$$

Agreement without compromise If no keys were compromised, then a session event by I implies a matching session event by R, with agreement on the nonce, proof, and commitment. This ensures that the protocol is correct in the uncompromised setting. Since commitments are unique (as proven in one of the helper lemmas), this also implies injective agreement.

$$\forall keys, data, t_1 \,.\, \mathtt{session}(\text{'I'}, keys, data) \ @\ t_1 \wedge \neg\,(\exists k, t_c \,.\, \mathtt{compromise}(k) \ @\ t_c) \implies \exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{session}(\text{'R'}, keys, data) \ @\ t_0.$$

Guaranteed compromise detection of a key in prior sessions Assuming at most one key is compromised, if I completes a session at t_3, then either every prior session completed by I has a corresponding session completed by R, or I will detect key compromise. Formally,

$$\begin{aligned}\forall pk_I, pk_R, data_1, data_2, t_1, t_2, t_3 \,.\, t_1 \lessdot t_2 \lessdot t_3 \wedge \mathtt{session}(\text{'R'}, \langle pk_I, pk_R\rangle, data_2) \ @\ t_2 \wedge \\ \mathtt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_2) \ @\ t_3 \wedge \mathtt{session}(\text{'I'}, \langle pk_I, pk_R\rangle, data_1) \ @\ t_1 \wedge \\ \neg\,(\exists t_c, t_{c2}, k_R, k_I \,.\, pk_R = \mathsf{pk}(k_R) \wedge pk_I = \mathsf{pk}(k_I) \wedge \\ \mathtt{compromise}(k_R) \ @\ t_c \wedge \mathtt{compromise}(k_I) \ @\ t_{c2}) \implies \\ (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{session}(\text{'R'}, \langle pk_I, pk_R\rangle, data_1) \ @\ t_0) \vee (\mathtt{detect}(pk_I, pk_R) \ @\ t_3).\end{aligned}$$

Guaranteed compromise detection of the responder's key in future sessions If there was a previous session with agreement and the adversary has not revealed R's state since that session, then a session completed by I will either have a matching session by R or I will detect that there is not. Formally,

$$\begin{aligned}\forall pk_I, pk_R, data_1, data_2, t_0, t_1, t_3 \,.\, t_0 \lessdot t_1 \lessdot t_3 \wedge \\ \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_0 \wedge \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_1 \wedge \\ \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_3 \wedge \neg(\texttt{detect}(pk_I, pk_R) \mathbin{@} t_3) \wedge \\ \neg\,(\exists t_c, x \,.\, (t_0 \lessdot t_c \lessdot t_3) \wedge \texttt{compromiseState}(\langle pk_I, pk_R\rangle, x)) \implies \\ (\exists t_2 \,.\, t_2 \lessdot t_3 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_2)\,,\end{aligned}$$

4.3.2 Modifying the ISO-IEC 9798-3-3 protocol

The ISO-IEC 9798 standard defines a family of entity authentication protocols, ranging from one- to five-pass symmetric and asymmetric key based authentication protocols. These protocols are not numbered directly, but Basin *et al.* provide a useful nomenclature in [3] from which we refer specifically to the 9798-3-3 protocol. The 9798-3-3 protocol is a two-pass mutual authentication protocol using digital signatures, and its simplicity makes it straightforward to use in settings where secrecy is not required and some kind of public key infrastructure exists, as in the case of smart card authentication. As such, we use it as an example of a generic smart card authentication protocol which may be used to authenticate to a wide range of different services, as with the Common Access Card. Note that in [3], Basin *et al.* demonstrate that the 9798-3-3 protocol is vulnerable to a reflection attack if an agent may take on both roles; in the following we assume that the reader and card (the initiator and responder respectively) are always assigned different signing keys, preventing this attack. This is typical in real deployments, where the key used by the card is generated on the card or during manufacturing, to prevent keys from being exfiltrated from the card.

Smart card authentication is used in a number of domains, some with millions of users each with their own unique card. These cards are used to authenticate access to, for example, buildings, credit accounts, networks, or online services. Card cloning is an occasional concern, either because of poor cryptography [15], or various side channel attacks allowing extraction of keys, like differential power analysis [31,34] and—at much greater expense—electron microscopy combined with chip probing [19]. Most smart card security is built around a substantially increased cost for an attacker to clone a card, and infeasibility of cloning on a large scale, rather than an assumption that it can protect against a targeted attack with physical access to the card.

In the United States, the Common Access Card is used for authentication in all of the domains listed above, and is the standard identification for all active-duty defence personnel. The security-critical nature of military resources makes targeted cloning attacks a more direct threat. The scale of the system even makes attacks by card suppliers a concern, as a supplier could build in a method to ex-filtrate keys or stockpile them at the time of manufacture. Rapid detection of cloned cards, even at the time they are first used, would provide additional assurance on top of existing technology.

In Figure 8, we present a modified 9798-3-3 protocol that provides strong detection properties. We show that it is possible for a smart card authentication protocol to not only swiftly detect and revoke cloned cards, but also invalidate any previously 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 if the adversary can also compromise the key of the reader, *and* intercept messages between the card and the reader. We briefly describe the protocol flow here before formalizing these properties in the following section.

At the initialization phase, each CAC stores a unique commitment com_0 created by the back-end server. In the i-th session, when a reader detects a CAC, the reader communicates with the back-end server, which generates a message m_1 signed with the server's private key $\mathsf{sk}(I)$ containing the identities of the server and card, a fresh nonce ni_i, and the previous commitment provided by the card com_{i-1}. The reader forwards this to the card.

The card verifies the signature, and that the provided com_{i-1} agrees with its local memory. If these verifications succeed, the CAC generates a new nonce nr_i and a pair (sk_i, pk_i) of signing and verification keys. The CAC creates a new commitment com_i using sk_i, signs com_i again using sk_{i-1}, and uses its long-term key $\mathsf{sk}(R)$ to create a signed message m_2 from the identities (R, I), the two nonces (nr_i, ni_i), the signed com_i, and the public key that verifies the previous commitment com_{i-1}. The CAC then sends this message to the reader to be forwarded back to the server. The server can check that the message contents are correct, and then verifies the signatures of the commitments com_{i-1} and com_i against the provided proof pk_{i-1}, raising a detection alert if this validation fails; if it succeeds, both the back-end server and CAC update their state. A detection event can immediately revoke the card's access and trigger the relevant remediation procedure; otherwise the card has successfully authenticated and the appropriate action can follow (e.g. opening the door).

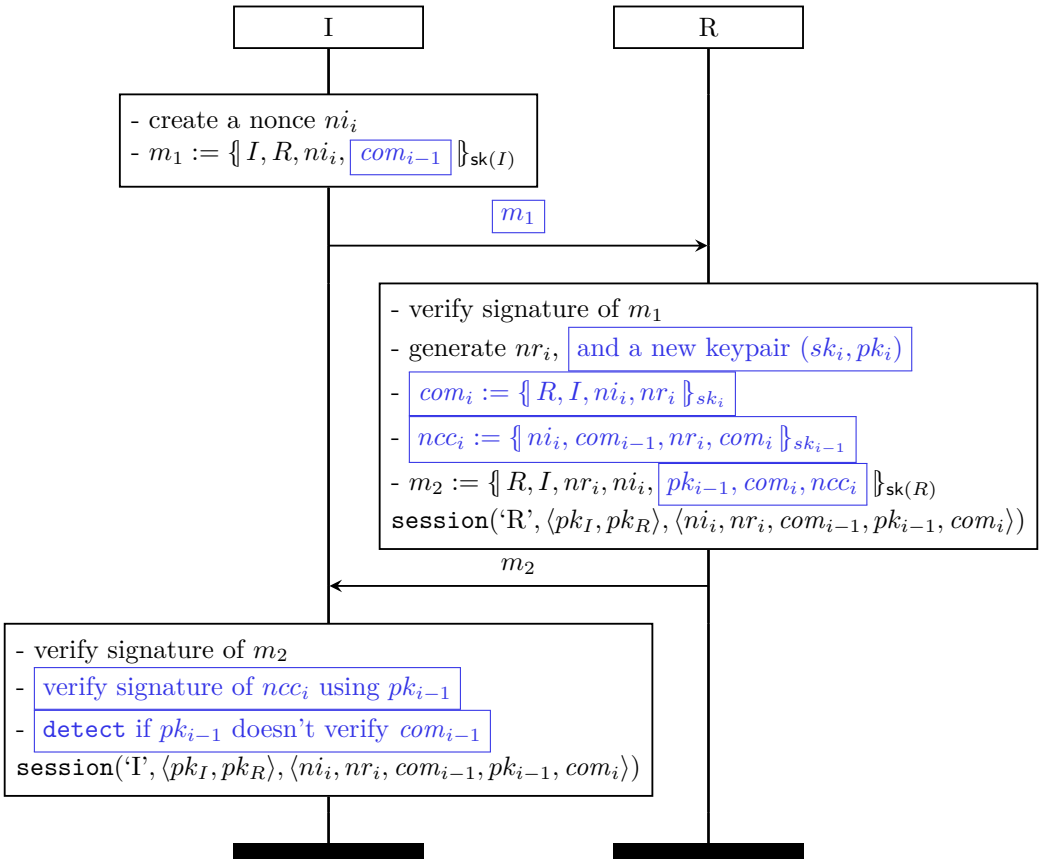

Figure 8: The i-th session of the modified ISO-IEC 9798-3-3 standard protocol. Modifications are boxed and highlighted in blue.

In this scenario, the modified protocol provides both detection of acausal action and contradicting state changes 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 internal 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.

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 can be used for these to reduce the computation required by the card.

As with the commitment-based protocol example above, we augment our adversary with the ability to compromise agent state, labelled with `compromiseState`. The modified ISO-IEC 9798-3-3 has the following security properties.

Sound detection If there is a `detect` event triggered for a particular key pairing, then the responder's key was compromised before the `detect` event. Formally,

$$\forall k_I, k_R, t_1 \,.\, \mathtt{detect}(\mathsf{pk}(k_I), \mathsf{pk}(k_R)) \mathbin{@} t_1 \implies (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{compromise}(k_R) \mathbin{@} t_0)\,.$$

Agreement without compromise If no keys were compromised, then a session event by I implies a matching session event by R, with agreement on the nonce, proof, and commitment. This ensures that the protocol is correct in the uncompromised setting. Since commitments are unique (as proven in one of the helper lemmas), this also implies injective agreement.

$$\forall keys, data, t_1 \,.\, \neg(\exists k, t_c \,.\, \mathtt{compromise}(k) \mathbin{@} t_c) \wedge \mathtt{session}(\text{'I'}, keys, data) \mathbin{@} t_1 \implies \exists t_0 \,.\, t_0 \lessdot t_1 \wedge \mathtt{session}(\text{'R'}, keys, data) \mathbin{@} t_0.$$

Guaranteed compromise detection of a key in prior sessions Assuming at most one key is compromised, if I completes a session at t_3, then either every prior session completed by I has a corresponding session completed by R, or I will detect key compromise. Formally,

$$\begin{aligned}\forall pk_I, pk_R, data_1, data_2, t_1, t_2, t_3 \,.\, t_1 \lessdot t_2 \lessdot t_3 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_2 \wedge \\ \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_3 \wedge \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_1 \wedge \\ \neg\,(\exists t_c, t_{c2}, k_R, k_I \,.\, pk_R = \mathsf{pk}(k_R) \wedge pk_I = \mathsf{pk}(k_I) \wedge \\ \texttt{compromise}(k_R) \mathbin{@} t_c \wedge \texttt{compromise}(k_I) \mathbin{@} t_{c2}) \implies \\ (\exists t_0 \,.\, t_0 \lessdot t_1 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_0) \vee (\texttt{detect}(pk_I, pk_R) \mathbin{@} t_3).\end{aligned}$$

Guaranteed compromise detection of the responder's key in future sessions If there was a previous session with agreement and the adversary has not revealed R's state since that session, then a session completed by I will either have a matching session by R or I will detect that there is not. Formally,

$$\begin{aligned}\forall pk_I, pk_R, data_1, data_2, t_0, t_1, t_3 \,.\, t_0 \lessdot t_1 \lessdot t_3 \wedge \\ \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_0 \wedge \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_1) \mathbin{@} t_1 \wedge \\ \texttt{session}(\text{`I'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_3 \wedge \neg(\texttt{detect}(pk_I, pk_R) \mathbin{@} t_3) \wedge \\ \neg\,(\exists t_c, x \,.\, (t_0 \lessdot t_c \lessdot t_3) \wedge \texttt{compromiseState}(\langle pk_I, pk_R\rangle, x)) \implies \\ (\exists t_2 \,.\, t_2 \lessdot t_3 \wedge \texttt{session}(\text{`R'}, \langle pk_I, pk_R\rangle, data_2) \mathbin{@} t_2)\,,\end{aligned}$$

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, 25, 38, 39] for detecting cloned RFID tags in the supply chain have been widely studied. In [39], 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 [29] 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, 18, 21]) 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) [32] 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 [13, 14], Cohn-Gordon, Cremers, and Garratt introduced *Post-Compromise Security* (PCS): security guarantees for communication after a party's long-term keys are compromised. This can be 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 [23] 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. A conceptual difference is that they focus on misbehaving parties (for example, election authorities that are expected to behave in a certain way, but might not do so). In contrast, we focus on compromised parties, whose key material is in the possession of both the party and the adversary.

6 Conclusions

We have described and explored designs for protocols that detect when an adversary uses an agent's compromised secrets. Our design principles and constructions directly led to suggesting improvements for many deployed systems, enabling them to automatically detect the compromise 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.

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. The authors would like to thank Benedikt Schmidt for suggesting the Keyless SSL case study as an application of our ideas, as well as the reviewers of CSF 2017 for their helpful comments. Mark Ryan's contribution was part-funded by HP Inc.

References

[1] Martín Abadi and Roger M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[2] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack resilient public-key infrastructure. In *Proc. of the ACM Conference on Computer and Communications Security*, CCS '14, pages 382–393, New York, NY, USA, 2014. ACM.

[3] David A. Basin, Cas J. F. Cremers, and Simon Meier. Provably repairing the ISO/IEC 9798 standard for entity authentication. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2012.

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

[5] Erik-Oliver Blass, Kaoutar Elkhiyaoui, and Refik Molva. Tracker: Security and privacy for RFID-based supply chains. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2011.

[6] Kevin Bowers, Ari Juels, Ronald L. Rivest, and Emily Shen. Drifting keys: Impersonation detection for constrained devices. In *Proc. IEEE INFOCOM*, pages 1025–1033, April 2013.

[7] Richard S. Canter. Lock with security counter, February 1974. US Patent 3,789,639.

[8] Michael R Carr. Smart card technology with case studies. In *Proc. International Carnahan Conference on Security Technology*, pages 158–159. IEEE, 2002.

[9] Melissa Chase and Sarah Meiklejohn. Transparency overlays and applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 168–179, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Laurent Chuat, Pawel Szalachowski, Adrian Perrig, Ben Laurie, and Eran 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.

[11] Cisco. The zettabyte era: Trends and analysis, 6 2017.

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

[13] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On Post-compromise Security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, 2016.

[14] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. Post-compromise security. Cryptology ePrint Archive, Paper 2016/221, 2016.

[15] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A practical attack on the MIFARE classic. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2008.

[16] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[17] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, Santa Clara, CA, March 2016. USENIX Association.

[18] Johan Håstad, Jakob Jonsson, Ari Juels, and Moti Yung. Funkspiel schemes: An alternative to conventional tamper resistance. In *Proc. of the ACM Conference on Computer and Communications Security*, CCS '00, pages 125–133, New York, NY, USA, 2000. ACM.

[19] Clemens Helfmeier, Dmitry Nedospasov, Christopher Tarnovsky, Jan Starbug Krissler, Christian Boit, and Jean-Pierre Seifert. Breaking and entering through the silicon. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 733–744. ACM, 2013.

[20] ISO/IEC. Entity authentication – Part 3: Mechanisms using digital signature techniques. Standard ISO/IEC 9798-3:1998, International Organization for Standardization, 1998.

[21] Gene Itkis. Cryptographic tamper evidence. In *Proc. of the ACM Conference on Computer and Communication Security*, pages 355–364. ACM Press, 2003.

[22] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan 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.*, pages 279–296, 2016.

[23] Ralf Küsters, Tomasz Truderung, and Andreas 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*, pages 526–535, 2010.

[24] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[25] Mikko Lehtonen, Florian Michahelles, and Elgar Fleisch. How to detect cloned tags in a reliable way from incomplete RFID traces. In *RFID, 2009*, pages 257–264. IEEE, 2009.

[26] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc.*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[27] Kevin Milner. *Detecting the misuse of secrets: foundations, protocols, and verification.* PhD thesis, University of Oxford, 2018.

[28] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. Tamarin models regarding compromise detection, 2026. `https://people.cispa.io/cas.cremers/downloads/models/compromise-detection/tamarin-examples-detection.zip`.

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

[30] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[31] David Oswald and Christof Paar. Breaking mifare desfire MF3ICD40: power analysis and templates in the real world. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2011.

[32] Trusted Computing Group. TPM 1.2 Specification, July 2014. `https://www.trustedcomputinggroup.org/tpm-1-2-protection-profile/`.

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

[34] Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 Seconds: Hijacking with Hitag2. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 237–252. USENIX Association, 2012.

[35] John Wilkes. Detector lock with key (c. 1675 – c. 1700). Rijksmuseum, Object BK-NM-886.

[36] Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: A new formalized PKI with verifiable trusted parties. *Comput. J.*, 59(11):1695–1713, 2016.

[37] Jiangshan Yu, Mark Ryan, and Cas Cremers. DECIM: Detecting Endpoint Compromise In Messaging. *IEEE Trans. Inf. Forensics Secur.*, 13(1):106–118, 2018.

[38] David Zanetti, Leo Fellmann, and Srdjan Capkun. Privacy-preserving clone detection for RFID-enabled supply chains. In *RFID, 2010*, pages 37–44. IEEE, 2010.

[39] Davide Zanetti, Srdjan Capkun, and Ari Juels. Tailing RFID tags for clone detection. In *20th Annual Network and Distributed System Security Symposium, NDSS*, 2013.

A Overview of Major Changes

v2.0, 21 Jan 2026

Completely revised the core terminology of the paper.

Previously, the core terminology of the paper was to "detect the *misuse* of secrets". This is factually accurate: if a secret is compromised but never used, then we cannot detect; thus, the best we can hope for is to detect the misuse of compromised secrets.

However, in the years after we first wrote our paper, the terminology of "compromise" gained more and more traction in this domain. Thus, several papers were written using the compromise-based terminology that missed our initial observations: our original paper's terminology did not properly connect to the modern developments around secure messaging, ratcheting, and key/certificate transparency.

Thus, we decided to reformulate the entire paper in terms of "detecting compromised secrets", instead of the previous "detecting the misuse of secrets". We did not change any of the results: only the terminology changed.

We performed minor maintenance on the Tamarin models to bring them in line with more modern Tamarin features and conventions.

v1.1, 4 April 2019

Many improvements, added examples, and additional related work.

v1.0, 11 March 2017

Initial release, entitled "Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Automation". This version corresponds to the IEEE CSF 2017 paper of the same title.