Insider Threats and Auctions: Formalization, Mechanized Proof, and Code Generation

Florian Kammüller[1]*, Manfred Kerber[2], and Christian W. Probst[3]
[1]*Middlesex University, The Burroughs London NW4 4BT, United Kingdom*
F.Kammueller@mdx.ac.uk
[2]*University of Birmingham, Edgbaston Birmingham B15 2TT, United Kingdom*
M.Kerber@cs.bham.ac.uk
[3]*Technical University of Denmark*
Anker Engelunds Vej 1 Bygning 101A 2800 Kgs. Lyngby, Denmark
cwpr@dtu.dk

Abstract

This paper applies machine assisted formal methods to explore insider threats for auctions. Auction systems, like eBay, are an important problem domain for formal analysis because they challenge modelling concepts as well as analysis methods. We use machine assisted formal modelling and proof in Isabelle to demonstrate how security and privacy goals of auction protocols can be formally verified. Applying the costly scrutiny of formal methods is justified for auctions since privacy and trust are prominent issues and auctions are sometimes designed for one-off occasions where high bids are at stake. For example, when radio wave frequencies are on sale, auctions are especially created for just one occasion where fair and consistent behaviour is required. Investigating the threats in auctions and insider collusions, we model and analyze auction protocols for insider threats using the interactive theorem prover Isabelle. We use the existing example of a fictitious cocaine auction protocol from the literature to develop and illustrate our approach. Combining the Isabelle Insider framework with the inductive approach to verifying security protocols in Isabelle, we formalize the cocaine auction protocol, prove that this formal definition excludes sweetheart deals, and also that collusion attacks cannot generally be excluded. The practical implication of the formalization is demonstrated by code generation. Isabelle allows generating code from constructive specifications into the programming language Scala. We provide constructive test functions for cocaine auction traces, prove within Isabelle that these functions conform to the protocol definition, and apply code generation to produce an implementation of the executable test predicate for cocaine auction traces in Scala.

Keywords: Insider Threats, Auctions, Formal Methods, Code generation

1 Introduction

We provide a logical reasoning framework for insider threats of auctions by combining earlier work on the formal modelling and analysis of auctions [1], the Isabelle insider framework [2], as well as a framework for the modelling and analysis of security protocols using Isabelle's inductive definitions [3]. Since these previous works are all based on the Isabelle theorem prover, the integration of these techniques naturally provides a single framework for the rigorous mathematical modelling and machine supported proof of auction insider threats.

When eBay became a popular internet phenomenon, Stajano and Anderson [4] raised the question of privacy and anonymity issues by publishing a fictitious cocaine auction protocol that maximizes the trust

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 8:1 (Mar. 2017), pp. 44-78
*Corresponding author: Middlesex University, Faculty of Science and Technology Town Hall, The Borroughs NW4 4BT London, United Kingdom, Tel: +44-2084-114930

issues apparent in eBay. In their paper, they further propose physical radio broadcast as an anonymous implementation for this auction. We provide instead an anonymity layer based on cryptography and adapt it to public and private key pairs instead of the originally suggested Diffie-Hellman key establishment. Inspired by this protocol, we use it as a running example to exhibit major insider threats in auctions. We first show that the so-called sweetheart deal can be excluded. A sweetheart deal prevents participants from having a real chance in the auction because the seller and one of his friends have already agreed before the auction that this friend, the sweetheart, will get the goods. The auction is merely used to determine the price. In our approach, we can formally specify the protocol using an inductive definition and proof within Isabelle interactively that no sweetheart deal is possible in faithful implementations. Another insider threat to auction is the collusion of the bidders, also known as "ringing". The collusion of bidders leads to the seller only getting the lowest bid, the "reserve price". To model this attack, we can employ concepts from the Isabelle insider framework [2] that enable expressing the impersonation of actors by others and extend it by a notion of rational agent, the "homo oeconomicus" assumption. Given these, we can formally prove that collusions are always possible, that is, the bidder may only get the reserve price.

To test our formal Isabelle based analysis techniques, we propose a formalization, implementation, and security analysis of the cocaine auction protocol within the interactive theorem prover. Furthermore, we can show the practical value of the approach by formally refining the specification into a constructive one from which we can then generate executable code for the testing of cocaine auctions.

In summary, this paper provides the following technical contributions: (a) a formalization of the cocaine protocol using Isabelle's inductive approach including the formalization and proof of the absence of the sweetheart deal and the impossibility of excluding collusion of insiders; (b) the extension of the inductive approach to auctions by expressing arbitrary numbers of rounds, broadcast messages, an anonymity layer, and by merging with the Isabelle insider framework; (c) a practical solution by defining a constructive test predicate that implements the protocol, applying the code generation mechanism of Isabelle to generate Scala code from that, and proving correctness of the test predicate with respect to the specification within Isabelle. These technical contributions lead to a deeper understanding of the relationship between auctions and security protocols with regard to insider threats paving the way for a more substantial Isabelle insider framework integrating relevant parts of the inductive approach to model and verify auction systems in the presence of insider threats. This paper is an extension of a previously published workshop paper [5]. It is substantially extended by the above construction (c) which is mainly contained in Section 6 and the Appendix.

Overview: Section 2 first reviews the auction literature focusing on security attacks involving collusions. Furthermore, we summarize how the insider threat literature handles the collusion of insiders. Section 3 introduces the cocaine auction protocol as our running example, before we present its formalization in the Isabelle inductive approach in Section 4. Next, we discuss to what extent the formal approach is useful to express possible insider threats to auctions, whether the threats exhibited in the case study are representative and complete. The extension by the constructive test predicate, code generation, and correctness proofs are subject of Section 6 and the generated code is contained in the Appendix. To finalize, we summarize challenges for future research (Section 7).

2 Auction Attacks and Colluding Insiders

In this section, we want first to give a very brief introduction to auctions and then discuss one of the biggest problem of auctions, collusion.

Auctions come in different forms, for instance, there are so-called first price and second price auctions. In first price auctions, the winner is the bidder with the highest bid and has to pay the value of

his or her bid[1]. In second price auctions, the winner is again the bidder with the highest bid, but has to pay the value of the second highest bid. The latter is in some ways more complicated than a first price auction.[2] So why is there interest in second price auctions? Since it makes matters much easier for the bidders. Vickrey's theorem states that in a second price auction a bidder cannot do better than bidding what the object is actually worth to her. If a bidder bid more than the object is worth to her she would make a loss on winning. If she bid less than the object is worth to her then potentially she would not realize a gain. If, however, she bids exactly how much she values the object all this cannot happen.

In addition to the distinctions between first price and second price auctions, auctions come in single round auctions, or auctions with several rounds (either ascending or descending). New auctions are still designed, for instance, recently for mobile phone licenses in different countries.

According to one of the leading auctions designers [6, p.152] "the two issues that really matter [in auction design] are attracting entry and preventing collusion." The first issue is that if not sufficiently many agents attend an auction then this is bad news for the seller. In the preface, Klemperer mentions the case of "a German auction of three blocks of spectrum [for which only three bidders had turned up and] which therefore sold only at a tiny reserve price." The second issue is more interesting for the purpose of this paper (albeit related to the first. If the bidders collude it only looks as if there were many bidders, but actually all the colluding bidders should count only as one). [6, p.152]: "Ascending auctions allow bidders to use the early rounds to signal each other how they might 'collusively' divide the spoils, and if necessary, use later rounds to punish any rivals who fail to cooperate.[...] By contrast, a (first-price) *sealed-bid auction* provides no opportunity for either signalling or punishment to support collusion."

While it is not possible to send signals, it is still possible to collude in such an auction as well, be it a first price or second price auction.

Krishna [7, p.152] describes collusion in second price auctions. Assume you have a number of agents and a bidding ring (or cartel) among those. The cartel would determine the one among them who values the object most and only the high bidder would submit a serious bid, all the others would either submit nothing or something that is so low that there is no danger that it will bring the price up. The bidders outside the cartel are not affected by this. 'A bidding ring generates profits for its members, of course, by suppressing competition.' In the extreme case all bidders are part of the cartel and the object will be sold at the reserve price at the expense of the seller, who without the existence of the cartel would achieve a higher price.

A separate issue does occur in a scenario in which the auctioneer cannot be trusted, for instance, since he is on the side of one of the participants and abuses his privileged position, for instance, to provide information to some participants but not to others.

Vice versa within the insider threat community, the collusion of insiders has been recognized as a main pattern of insider threats. The CMU-CERT Insider Threat Guide [8] names the *Ambitious Leader* pattern as one of the four main patterns of insider threats. This pattern describes an outsider – the ambitious leader – that works together with (at least) two insiders in separate infrastructures thereby realizing an attack that would not normally be possible for any of the involved insiders on their own. This pattern is a collusion of insiders. We used this pattern to show that the Isabelle insider framework [2] is capable of expressing all known insider threats.

[1] In case of two or more bidders with the same highest bid certain tie breaking rules apply.

[2] In case of combinatorial auctions when several goods are auctioned at the same time, the determination of the winners and the prices to pay may be computationally very complex.

Figure 1: Illustration of the cocaine auction as imagined by the first author.

3 Cocaine Auction Protocol

In this section we will first summarize the cocaine protocol as described in [4], then look at potential formalizations, formalize the protocol, and discuss possible attacks.

3.1 Protocol

"Several extremely rich and ruthless men are gathered around a table. An auction is about to be held in which one of them will offer his next shipment of cocaine to the highest bidder. The seller describes the merchandise and proposes a starting price. The others then bid increasing amounts until there are no bids for 30 consecutive seconds. At that point, the seller declares the auction closed and arranges a secret appointment with the winner to deliver the goods" [4] (see Figure 1). This is the short introduction to the cocaine auction protocol given in the original paper. This example serves as a model for eBay-like auctions where trust is an issue. In the eBay model, the auction house could drive up the sale price since it asks the bidders to reveal their maximum amount they are prepared to pay. The users simply have to trust that eBay will not exploit that knowledge to drive up the price (which would be profitable for the auction house because it takes a commission which is a percentage of the sale's price). The eBay "peer review" system in which users give each other reliability ratings has proven to be a quite successful method to guarantee trust between sellers and buyers. However, trusting the auction house remains a problem. The cocaine auction protocol has been designed as an "exaggerated case that makes the trust issue unambiguous" [4]. There are several assumptions imposed on the cocaine auction protocol in order

to minimize trust.

- Nobody trusts anybody else more than is strictly necessary.
- The people that take part in the auction all know each other (otherwise one of them could be a police agent).
- Nobody that makes a bid wants to be identified to the other bidders nor to the seller.
- Nobody apart from the seller and the buyer should know who won the auction; even the seller should only find out the identity of the buyer when committing to the sale, that is, at the time of exchanging the goods at the secret appointment.
- Nobody of the participants should have to trust any of the other participants; in particular there should not be an independent judge or policeman. The protocol must be self-enforcing.

3.2 Possible Implementations

For the context of this paper, we just assume an *anonymous broadcast:* a mechanism for broadcasting messages to participants without revealing the identity of the sender. This represents an anonymity layer that can be implemented by cryptographic techniques, for example, using the dining cryptographers algorithm [9], or by using physical broadcast short-range radio networking facility, e.g., Piconet. In fact, the latter possibility is the main point of Stajano's and Ross' paper [4] to advocate the use of physical broadcast to implement the anonymity layer. For the context of this paper, we abstract from the concrete implementation of this anonymity layer. In the formal description of the protocol in Section 4, we will instead rely on the inductive approach to protocol verification and use address spoofing as a means for the senders to hide their identity from the receivers. There are two important details that we need to keep in mind when considering the practical implementation of the protocol.

1. The seller needs a mechanism to identify the winner.
   - A potential problem with this is that anyone can come later and claim to have said "yes" (that is, made a bid) in the winning round.
   - A solution to this is that such a "yes" message (bid) contains a one-way function of a secret nonce.
   - The seller will ask the winner to exhibit the original nonce.
2. At finish of the auction, the seller prefers to give a secret appointment to the winner.
   - "See you on Tuesday at 06:30 in the car park of Heathrow terminal 5" (rather than exchanging suitcases of cocaine for cash under the noses of all the losing bidders).
   - On the other hand, the identity of the buyer should not be revealed to the seller until the latter commits to the sale (in order to protect the winner from not getting the bid for other biases, for instance, since he is from the "wrong family").

3.3 Protocol in Alice-Bob notation

Assuming an anonymity layer in a first approximation, the protocol can be described as follows:

- The identity of the seller is known to buyers.
- Buyers' messages are anonymous; seller's are not; all messages are broadcast.

- The protocol is a succession of rounds i of simple bidding.
  - The seller announces bid price b_i of round i.
  - Buyers have up to 30 seconds to say "yes".
  - As soon as a buyer says "yes", he is winner of the round, w_i.
  - A new round starts.
  - If 30 seconds elapse in round i with no bid, winner of the auction is w_{i-1}.

The implementation of this protocol is given informally [4] based on the use of the Diffie-Hellman key-establishment algorithm [10]. For convenience, we briefly summarize the main idea of the Diffie-Hellman key-establishment algorithm here. This algorithm uses a prime number modulus p and a generator $g \in \mathbb{Z}_p$ known to sender and receiver A and B. In addition, A keeps a secret number $a \in \mathbb{N}$ and B a secret number $b \in \mathbb{N}$. The algorithm works in two phases, establishing the shared secret $g^{ab}\texttt{mod}\ p$ between A and B without them ever exchanging their secrets a and b in clear. In the first phase, A calculates $g^a\texttt{mod}\ p$ and sends it to B; B calculates $g^b\texttt{mod}\ p$ and sends it to A. It is computationally unfeasible for large prime numbers p to get a or b from these because of the Discrete-Logarithm-problem. In the second phase, A calculates $(g^b\texttt{mod}\ p)^a\texttt{mod}\ p$; B calculates $(g^a\texttt{mod}\ p)^b\texttt{mod}\ p$. Now, both have the shared secret because modular arithmetic gives

$$(g^b\texttt{mod}\ p)^a\texttt{mod}\ p = g^{ba}\texttt{mod}\ p = g^{ab}\texttt{mod}\ p = (g^a\texttt{mod}\ p)^b\texttt{mod}\ p.$$

The security of the algorithm depends on the high complexity of calculating $g^{ab}\texttt{mod}\ p$ given g, p and the sent messages $g^a\texttt{mod}\ p$ and $g^b\texttt{mod}\ p$. This problem is known as the Diffie-Hellman-problem and seems intuitively related to the computationally intractable Discrete-Logarithm-problem. The Diffie-Hellman-problem has been proved to be equivalent (for certain cases) to the Discrete-Logarithm-problem [11].

To formalize the protocol in the semi-formal "Alice-Bob" notation we first give the assumptions.

- Generator g and modulus p are public auction parameters.
- Anonymous "yes" message of winner w_i is g^{x_i}.
- Seller uses his (random) secret y to send the secret appointment to final winner w_i encrypted with Diffie-Hellman key $g^{x_i y}$.
- Possible variants for disambiguation and conciseness are possible.
  - Succession of bid prices b_i is pre-specified (conciseness).
  - At beginning of round i, seller broadcasts the "yes" message $g^{x_{i-1}}$ of winner of previous round to arbitrate races.
  - Bidders should include the b_i in their "yes" messages.

Some extensions to the standard point-to-point messaging that is common in Alice-Bob-notation are needed to express the anonymous communication. Stajano and Ross introduce a dedicated notation, which we adapt here for simplicity slightly.

- $\mathscr{D}$ is the set of auction principals including the seller S with secret y.
- $?A_i$ represents an anonymous principal in round $i = 1, \ldots, n-1$ with secret number x_i.
- Winning round $n-1$.

The cocaine auction protocol can then be specified using Diffie-Hellman and the notations $\{a,b\}$ for message concatenation and $\{m\}_K$ for encryption of message m with key K.

0. $S \rightarrow \mathscr{D}$: $g^y \bmod p$

i. $?A_i \rightarrow \mathscr{D}$: $\{g^{x_i} \bmod p, b_i\}$

n. $S \rightarrow \mathscr{D}$: $\{b_i, \mathit{MeetingAppointment}\}_K$, $K = g^{x_{n-1}y} \bmod p$

3.4 Insider/Collusion Attacks

Stajano and Anderson state "[t]here are limits what can be achieved on the protocol level. It is always possible, [...], to subvert an auction when an appropriate combination of participants colludes against others". This collusion attack is known as "ringing". The colluding bidders can, for example, keep the price low and then share the profit. It is a strong statement that this is always possible but the statement is relative to the protocol level. In the formal modelling and analysis of the auction protocol using the inductive approach we will see that they are right. The challenge is to extend the usual protocol model with context information so that it becomes feasible to express this kind of collusion and consequently formalize and prove stronger security properties.

The second attack on the cocaine auction protocol is the so-called "sweetheart deal": the collusion between the seller and one of the bidders, that is, "seller not selling to the highest bidder" in [4, p. 4]. If this attack is attempted within the limits of the protocol, that is, the seller sends the secret appointment not to w_{n-1}, that is, the winner of the winning round $n-1$, but instead to one of the earlier bidders w_i for $i < n-1$, then the protocol in the above implementation with Diffie-Hellman fails. That is, if S sends message n encrypted with key $g^{x_i y} \bmod p$ instead of $g^{x_{n-1}y}$, the real winner w_i will not be able to decrypt the message. This failure of the protocol means that the attack is unfeasible if the protocol is implemented correctly. Formalizing the protocol should enable proving that this is the case. We will see how this can be formally proved even on our own implementation with public keys in Section 4.

The attacks on the cocaine auction protocol involve bidders and sellers. They are attacks on the auction that are only possible because they exploit privileges, like knowledge of keys, certificates, and access rights of roles, granted to peers in the protocol. Therefore, they can be categorized as insider attacks.

4 Formal Model

The Diffie-Hellman key exchange is a very efficient implementation of this protocol. However, we aim at using the established inductive approach to formally verify the protocol. Unfortunately, the inductive approach uses an abstract specification of symmetric or asymmetric keys and the Diffie-Hellman keys do not fall in those categories. In fact, the established Diffie-Hellman key $g^{x_i y} \bmod p$ is a symmetric key while the so-called ephemeral keys $g^y \bmod p$ and $g^{x_i} \bmod p$ are no encryption keys rather intermediate computations encrypting the secrets y and x_i for public exchange. Therefore, we provide here another possible implementation of the cocaine auction protocol using standard public-key cryptography.

The cocaine auction protocol can then be specified using the public key K_S of the seller S and its secret counterpart K_S^{-1} for decryption and public encryption keys K_{A_i} of the bidders with corresponding secret decryption keys $K_{A_i}^{-1}$.

0. $S \rightarrow \mathscr{D}$: K_S

i. $?A_i \rightarrow \mathscr{D}$: $\{K_{A_i}, b_i\}_{K_S}$

n. $S \rightarrow \mathscr{D}$: $\{b_{n-1}, \mathit{MeetingAppointment}\}_{K_{A_{n-1}}}$

This asymmetric version of the protocol works as follows.

- In Step 0, the seller sends to all bidders in the set $\mathscr{D}$ a public key K_S enabling them to send secret message to the seller.
- In each of the rounds i for $i = 1, \ldots, n-1$ the bidder sends anonymously using the sender address $?A_i$ his public key for the round K_{A_i} together with the prearranged bid b_i for the round encrypted with the public key of the seller K_S to the seller. The contents of this message are only visible to the seller since only he holds K_S^{-1}.
- In the final round (after the timeout has happened) the seller S broadcasts to all bidders in $\mathscr{D}$ the highest bid b_{n-1} and the secret message with the *MeetingAppointment*. This broadcast message is encrypted with the public key $K_{A_{n-1}}$ of the winner of round $n-1$ that the seller could retrieve from the message in the winning round $n-1$.

In comparison to the version using Diffie-Hellman key-exchange, this second implementation of the cocaine auction protocol seems slightly more complex. Although the latter implementation abstracts from a concrete public-key algorithm, Step i requires an encryption of the entire message $\{K_{A_i}, b_i\}$, whereas the Diffie-Hellman version requires only the computation of one ephemeral key $g^{x_i} \bmod p$. For example, if we consider RSA to be the concrete algorithm used in the second public key version, the key length, that is, the size of the modulus p would be roughly the same for RSA and Diffie-Hellman for equal strengths of security. At a closer look, however, computing $g^{x_i} \bmod p$ corresponds roughly to the same computation effort as computing $(\#\{K_{A_i}, b_i\})^{K_{A_i}} \bmod n$ (following the RSA-algorithm [12] where the # is the transformation of the message into a number for exponentiation with the RSA encryption key K_{A_i} modulo the public modulus $n = p * q$). The two implementations, or more precisely Steps i are of similar complexity, because, the ps and qs are of similar size (currently 1024 bits are still considered safe, although 2048 are recommended after the successful factorization of a 1024 prime equivalent to breaking of 700 bit RSA key and the Logjam attack on Diffie-Hellman [13]).

4.1 Isabelle's Inductive Approach to Security Protocol Verification

The interactive theorem prover Isabelle/HOL [14] implements classical higher order logic (HOL) for the modelling of application logics. Inductive definitions and datatype definitions can be written in a way close to programming languages. Semantic properties over datatypes can be formalized in a simple equation style by primitive recursion and are strongly supported by automated proof procedures based on rewriting, automated simplification, as well as externally coupled dedicated provers.

The inductive approach to security protocol verification by Paulson [3], the designer of the Isabelle system, picked up on the hype generated by the earlier model checking approach to security by Lowe [15]. In comparison, the inductive approach is more laborious as it requires human interaction, but it is unrivalled in its expressiveness which allows proofs beyond the ones that are usually done in model checkers. Although proofs in Isabelle/HOL are not performed automatically but have to be provided by the user, the increased expressiveness allows modelling protocols less abstractly than in a model checker. A protocol's definition is given as an inductive definition of the set of all "traces" that are allowed by the protocol. A trace is a list of events representing the sending and receiving of messages that happen in a possible run of the protocol. The inductive definition defines all possible behaviours of a protocol as the minimal set of traces described by the inductive rules corresponding to the protocol's communication steps.

Paulson's inductive approach is only a starting point for the modelling of insider threats to auction protocols. We see in this paper that the classical inductive approach needs to be extended with concepts developed for the Isabelle insider framework [2] in order to fully support reasoning about insiders. Since an Isabelle framework, like the inductive approach to Security Protocol Verification, is nothing other than a number of theory files containing a set of tailor-made definitions and related theorems, it can be easily extended and also integrated with other approaches like the Isabelle insider framework [2].

For the sake of self-sufficiency, we briefly present the main features of Isabelle's inductive approach concentrating on the parts we use.

4.1.1 Cryptography, Keys, and Messages

Security protocol specifications are constituted as sequences of communication steps between principals possibly adding abstract cryptographic functionality. For example, in the so-called "Alice-Bob"-notation a typical protocol step like

$$A \mapsto B : \{M\}_K$$

would read as: "A sends to B message M encrypted by key K". The key could be specified more precisely as a symmetric key or the private K_A^{-1} or public K_A key of an agent A.

The function `invKey` maps a public key to its matching private key, and vice versa.

```
type synonym key = nat
consts invKey :: key ⇒ key
```

Nonces are a means to avoid replay attacks. A nonce is a large random number. A message that requires a reply can incorporate a nonce. The reply then must include that same nonce to prove that it is not a replay of a past message.

Protocol messages can then be defined as a recursive datatype `msg` building over the simple message constituents agents, numbers, nonces, and keys. Protocol messages usually consist of more than just one component. For the recursive cases, a `msg` can be a combination of other messages or a message encrypted with a key.

```
datatype msg = Agent agent
             | Number nat
             | Nonce nat
             | Key   key
             | MPair msg msg
             | Crypt key msg
```

Isabelle offers a sophisticated pretty printing syntax facility. This allows us to define the notation $\{x_1,\ldots,x_{n-1},x_n\}$ for the nested pairing `MPair` $x_1 \ldots (\texttt{MPair}\ x_{n-1}\ x_n)$ making the specifications of protocols very close to the on-paper notation.

The way datatypes are implemented in Isabelle provides the property that all of datatype constructors are injective functions. Therefore, the above definition `msg` implicitly entails the following theorem.

```
Crypt K M = Crypt K' M' ⟹ K = K' ∧ M = M'
```

This theorem says that a message `M` encoded with a key `K` yields only one ciphertext `Crypt K M` and no other message `M'` can be mapped onto this ciphertext – not even with a different key. The model is an oversimplification: in reality decryption with a wrong key `K'` would actually yield a result although quite likely pure rubbish. The oversimplification is justified as in reality checksums are introduced on the plaintext to exclude decryption with wrong keys.

4.1.2 Attacker Model, Events, and Traces

The principals are expressed by a datatype definition guaranteeing their distinctiveness. We assume a server, a number of friendly principals, and a spy. That is, in our model the attacker is explicitly modelled.

```
datatype agent = Server | Friend nat | Spy
```

The attacker can forge messages using all components he can derive from previous traffic. The inductive operators characterize the constituents of a protocol's messages (set `parts`), messages the attacker can extract from a protocol trace (set `analz`), and messages that the attacker can build (set `synth`).

Protocols are defined by inductive definitions describing the behaviour of principals taking part in the protocol. Behaviours are sets of possible event traces. A trace is a list of communication events, such as interleaved protocol runs.

Compared to the inductive definitions for `synth` and `analz`, protocol definitions are thus of a different type: rather than specifying a message set, they specify the behaviour of the communicating principals as traces of *events* defined as a datatype comprising different cases (represented as datatype constructors) of protocol communication events. The main case of an event is that an agent sends a message to another agent: the constructor `Says` takes three arguments of types `agent`, `agent`, and `msg` and returns one result of type `event`. The other constructors of the datatype `event` are `Gets` and `Notes` to specify the reception and storing of messages. Defining a protocol in the inductive definition consists of defining a set of traces of events representing all possible runs of the specified protocol. The attacker's behaviour is added by including `Fake` messages into traces. The analysis first derives the knowledge he can extract from the protocol (`analz`) and the messages he can synthesize (`synth`). This characterizes the attacker's behaviour and allows verification of security properties. Following the Dolev-Yao model [16], the `Spy` gets to know everything that is communicated along any channel. To this end, the inductive approach models a function `spies` that effectively deconstructs event traces into sets of messages.

We omit this definition because we concentrate on insider threats, that is, we cannot assume to have a clear distinction into "good" and "bad". Our attacker could be any agent. Besides insider threats, the cocaine auction protocol reveals other requirements to the inductive approach that go beyond classical security protocols.

1. In general, auctions necessitate an *arbitrary number of rounds*;
2. we need to represent *broadcast communication*;
3. we need to enable *anonymous sending* of messages.

4.2 Cocaine Protocol in Isabelle

Our formalization of the cocaine auction protocol resides in the theory file `CocaineAuction.thy` which is available online [17]. We provide next the inductive definition before we illustrate it by a simple example trace.

Formally, the inductive definition starts by introducing a constant `cocaine_auction`.

```
inductive_set cocaine_auction :: event list set
```

Following this introduction of the inductive set constant, a series of rules determines exactly which traces, i.e., lists of events, are in the set defining the semantics of the protocol. First, the rule `Nil` describes that the empty set is a possible trace, representing the beginning of each protocol run.

```
Nil: [] ∈ cocaine_auction
```

Similar to the specification of other protocols, we specify that Spy (see its specification in Message.thy [17]) can analyze and synthesize from what he "spies", that is, the set of things he knows (see also Event.thy [17]). Spy can then say all these things since he is an agent as well. The symbol $\Longrightarrow$ is the right associative implication of Isabelle's meta logic, that is, the first two sub-formulas below have to be read as a conjunction. The symbol # is the list constructor. Altogether, the following rule reads as "if a trace evsf is a (possible behaviour of a) cocaine auction *and* X is a message that can be synthesized from what can be analyzed from all the events in the trace that the spy can see, *then* a possible continuation is the sequence evsf extended by the event in which the spy utters to any principal RR this message X".

```
Fake:  evsf ∈ cocaine_auction  ⟹ X ∈ synth (analz (spies evsf))
       ⟹ Says Spy RR X # evsf ∈ cocaine_auction
```

Initially at the beginning of every cocaine auction, the Server (equal seller) sends his public key out to all agents that are present (all Friends). This definition uses list comprehension to produce a list of Says events for all i < friends setting it as the beginning of any run of the cocaine auction.

```
CA0: [Says Server (Friend i) (Key(pubK Server)). i ← [0..<friends]] ∈ cocaine_auction
```

In each round i, a Friend (bidder) can make an offer by saying "yes" corresponding to broadcasting a public encryption key encrypted with the Server key. Together with this public encryption key the bidder sends also the current price of that round (bid i) assumed to be given in advance by the function bid applied here to the round number i. We use the Isabelle specification device that allows to define an abstract function bid specifying that it should be injective, strongly monotonically increasing, and bid(0) = 0. A witness has to be given and the properties must be proved for a specification to be accepted by Isabelle. Authentication of the bidder is omitted here since later the bidder authenticates himself at the meeting point which he would not have been able to find without decrypting the message of the Server (see below – the final message of the Server is encrypted with the public key of the bidder transmitted here). The bidder uses the sender address Friend(friends) which is the "anonymous" address. That is, the bidders use "spoofing" to anonymize their messages. The public key they send is here formalized as pubK(Friend j) (see the theory file Public.thy [17]) but the owner of the key Friend j is assumed to be not visible – not even to the intended receiver of this broadcast message (the Server). The precondition on hd(evs) ensures that either

- this is the first round indicated by the last message (first in event list) being one of the initial messages of the Server with his pubK, or
- the previous event has been a message in which a bidder different from Friend j has made a bid and won the round. This is indicated in the last message being a bid similar to the current one but from Friend k with the previous bid(i - 1). Friend j now can increase the price to bid i.

```
CAi:  evs ∈ cocaine_auction ⟹ j <  friends ⟹
  hd(evs) = Says Server (Friend l)(Key(pubK Server)) ∧ i = 1  ∨
  hd(evs) = Says (Friend friends) Server
              (Crypt(pubK Server) {Key(pubK(Friend k)), Number(bid (i-1))}) ∧ i > 1
  ⟹ Says (Friend friends) Server
          (Crypt (pubK Server) {Key(pubK(Friend j)), Number(bid i)})
       # [Says (Friend friends) (Friend k)
          (Crypt (pubK Server) {Key(pubK(Friend j)), Number(bid i)})).
         k ← [0..<friends]]
       @ evs ∈ cocaine_auction
```

Timeout can take place at any time. We simply do not model time in the protocol and it is not necessary. Since we keep all possible traces as the semantics of a protocol in the inductive approach, any occurrence of timeouts is modelled. For the next rule of the auction protocol, we assume that timeout has just happened. Now, in this final round, the Server sends out the message with the secret appointment (encoded as a natural number for simplicity) and signs it with the public key of "some friend". This is the bidder that has won the previous round `n-1`. In this refined version, we enforce this by the precondition on `hd(evs)`. The winner of the previous round is represented in this most recent message by its public encryption key `pubK (Friend j)`. For the final message, this key of `Friend j` is chosen and the message with the secret appointment `mtng`[3] is sent encrypted with this public encryption key `pubK` of `Friend j` so that only he can decrypt it.

```
CAn: evs ∈ cocaine_auction ⟹
  evs = Says (Friend friends) Server
         (Crypt (pubK Server) {Key(pubK(Friend j)), Number(bid i)})
        # [Says (Friend friends)(Friend k){Key(pubK(Friend j)), Number(bid i)}.
           k ← [0..<friends]]
        @ evsf
  ⟹ [Says Server (Friend k)(Crypt (pubK(Friend j)){Number(bid i), Number mtng}.
       k ← [0..<friends]]
       @ evs ∈ cocaine_auction
```

This specification of the cocaine auction protocol establishes a few particular solutions extending the inductive approach to represent the peculiar requirements of the application (see previous section).

1. *Arbitrary numbers of rounds* in an auction are enabled and yet inconsistent traces are excluded since the rule `CAi` can be chained up any number of times but using a natural number counter `i` their interleaving can be controlled. The use of `i` and `i-1` is based on the mathematical library of Isabelle showing yet again the advantage of using this expressive, complete and consistent approach.

2. *Broadcast communication* is modelled explicitly using lists of messages to all principals. Again we see here the use of Isabelle libraries – this time for lists using the list comprehension in Haskell-like syntax: the formalization of a broadcast of a message `m` from a principal *A* to a community of principals $\mathscr{D}$ is `[Says A (Friend j) m.  j ←[0 ..<friends]`. This is simple and concise and corresponds quite closely to the specification using Alice-Bob notation as specified in the original paper [4] (also see Section 3.3).

3. *Anonymous sending* is implemented in our above protocol specification by *spoofing*. This term corresponds to a classical vulnerability of the TCP/IP protocol whereby the sender field in IP-packets can be freely replaced by an attacker to impersonate a principal. In order to hide his real identity, here in our inductive definition, the legitimate sender inserts (*spoofs*) the sender `Friend friends` in rule `CAi` using an identity that is out of bounds (only addresses strictly less than `friends` are admitted for `Friends`).

Although we stated above that we "abstract from a concrete implementation of the anonymity layer" the implementation by spoofing discussed in Point 3 comes very close to a technical solution of an anonymity layer. However, it does implicitly use the context assumptions of Paulson's inductive approach, here specifically, that keys remain unbroken, and that no attacker has a complete view of the network, but also others than the attacker can intercept, eavesdrop, and insert fabricated messages. These assumptions are

[3]The message is for simplicity embedded into the given message type `msg` provided in the inductive approach. It may be thought of as "encoded" as a number.

common as global assumptions for security protocol verification. They are mainly due to the Dolev-Yao model but are also inspired by common properties of the Internet protocol TCP/IP, like the spoofing property used. Clearly, in the context of insider threats we would need a slightly more global view as we consider not only the networking layer but also higher layers of infrastructures, like physical architectures, organizational policies, and even socio-technical system aspects [2].

4.2.1 Simple Example Trace

In order to illustrate the inductive definition of the cocaine auction protocol we consider here a simple example. Assume there are only 2 bidders, i.e., `friends = 2`. The following subset of traces of `cocaine_auction` step-by-step grows traces representing an auction in which each bidder makes just one offer before timeout appears after `Friend 1` bids finishing the auction. In the following set lists are postfixes of their successors, i.e., the traces repeat the previous trace. To make the exposition more succinct, we put "..." as much as possible omitting repetitions but their last element and highlighting the messages in the traces by different colors for each step of the protocol. There are precisely two rounds. We omit in particular all traces interleaved by `Fake` events.

```
{
[],
[ Says Server (Friend 0)(Key(pubK Server)), Says Server (Friend 1)(Key(pubK Server)) ],
[ Says (Friend friends) Server
       (Crypt(pubK Server) {Key(pubK(Friend 0)), Number(bid 1)}),
  Says (Friend friends) (Friend 0)
       (Crypt(pubK Server) {Key(pubK(Friend 0)), Number(bid 1)}),
  Says (Friend friends) (Friend 1)
       (Crypt(pubK Server) {Key(pubK(Friend 0)), Number(bid 1)}),
  Says Server (Friend 0)(Key(pubK Server)), Says Server (Friend 1)(Key(pubK Server)) ],
[ Says (Friend friends) Server
       (Crypt (pubK Server) {Key(pubK(Friend 1)), Number(bid 2)}),
  Says (Friend friends) (Friend 0)
       (Crypt(pubK Server) {Key(pubK(Friend 1)), Number(bid 2)}),
  Says (Friend friends) (Friend 1)
       (Crypt(pubK Server) {Key(pubK(Friend 1)), Number(bid 2)}),
  Says (Friend friends) Server
       (Crypt(pubK Server) {Key(pubK(Friend 0)), Number(bid 1)}),
  ... ],
[ Says Server (Friend 0) (Crypt(pubK(Friend 1)) {(Number(bid 2)), Number 42)},
  Says Server (Friend 1) (Crypt(pubK(Friend 1)) {(Number(bid 2)), Number 42)},
  Says (Friend friends) Server
       (Crypt (pubK Server) {Key(pubK(Friend 1)), Number(bid 2)}),
  ... ]
}
```

Clearly this is just an illustrative small example given by a selected subset of traces following the provided inductive rules in the given order `CA0, CAi, CAn`. The message that the Server sends in the final step encoding the meeting appointment as a number is randomly chosen to be 42; uniquely encoding a real appointment message like "meet me at 6.30am in the car park of Heathrow Terminal 5" would result in a much larger number.

4.3 Specification and Proof of Insider Threats

The insider attacks on auctions we investigate on the running example of the cocaine auction protocol are the sweetheart deal and the collusion of bidders known as "ringing" because they build a bidding ring

or cartel (see Section 2).

4.3.1 Sweetheart Deal

Our hypothesis is that the formal specification of a security protocol is sufficient to exclude the sweetheart deal. That is, the way we defined the rules for the cocaine auction should forbid that the seller announces the wrong bidder as the winner (his "sweetheart" – someone he has made a deal with outside the auction).

As a first observation, this attack is clearly an insider attack, as it it only possible because an insider – here the seller – colludes with another insider – a bidder. Together they use their privileges given by the policy – here, the auction – to achieve the attack goal – here, winning the auction.

The second observation is that the two final steps of the protocol – the way we defined it – prohibit that this insider threat may occur. In our formal specification, the second to last step of any protocol run (not counting interspersed Spy actions) is an application of `CAi` (see also the example given in the previous section to illustrate that point). The second to last message is a broadcast message of the bidder `Friend j` to the Server and all other bidders using the anonymous sender address `Friend friends` but containing an own public key `pubK(Friend j)` encrypted with the Server's public key. A list of events corresponding to this broadcast message must be starting the trace if the last rule `CAn` is invoked. When applying the rule, the last broadcast messages thus automatically use the key `(Crypt (pubK(Friend j))` for the encryption of the meeting appointment for that same `Friend j` as specified in the precondition. Therefore, no other `Friend k` for `j` $\neq$ `k` can be chosen by the Server.

Informally, this argument seems clear. But how can we prove this formally? The first step is the statement of the property which just formalizes the above observation. If any cocaine auction ends with a broadcast by the Server that the bidder `Friend j` is the winner, then the trace `evs` prior to this must have been a broadcast of this bidder. The additional assumption `0 < friends` in the theorem just excludes the empty set of bidders and generalizes the property for any finite number of bidders.

```
theorem no_sweetheart_deal:
0 < friends ⟹
[Says Server (Friend k) (Crypt (pubK (Friend j)) {Number(bid i), Number mtng}.
 k ← [0..<friends]]
@ evs ∈ cocaine_auction
⟹ ∃ evsf. evs =
  Says (Friend friends) Server
    (Crypt (pubK Server) {Key(pubK(Friend j)), Number(bid i)})
  # [Says Server (Friend k)
       (Crypt (pubK Server) {Key(pubK(Friend j)), Number(bid i)}).
    k ← [0..<friends]]
  @ evsf
```

Isabelle is an interactive theorem prover, that is, statements of theorems, like the above, need to be proved. This proof is supported by the fact that the rules for defining the protocol are an inductive definition. In Isabelle, and also in general, inductive definitions define the least set that is closed by a given set of rules. The principle of *rule inversion* allows us for a given element in this set to make a case analysis according to the cases defined by the rules of the inductive definition. In our case, the elements of the inductive set are traces, i.e., lists of events. Applying rule inversion technically in Isabelle is provided by the command `inductive cases` which provides us with a case analysis rule that reduces a property statement about trace sets of cocaine auctions, like the theorem `no_sweetheart_deal` to 6 subgoals corresponding to the premises of each of the rules of the inductive definition.

For the proof of the theorem, luckily, the first `empty` case is trivially true, while 4 other cases can be easily excluded. In Isabelle, elements of a datatype, here the datatypes of events, message, and agents,

are distinct if their arguments or constructors differ. Thus, two traces starting say with `Says Server X y` and `Says (Friend j) Z U` can never be equal because the arguments `Server` and `Friend j` are distinct therefore the application of constructor `Says` renders distinct elements. Consequently, the only real case that remains to be shown is the one that actually corresponds to the precondition of the theorem.

```
∃ evsf. evs =
  Says (Friend friends) Server
      (Crypt (pubK Server) {Key (pubK(Friend j)), Number(bid i)})
  # [Says (Friend friends)(Friend k)
       (Crypt (pubK Server) {Key(pubK(Friend j),Number(bid i)})).
     k ← [0..<friends]]
  @ evsf
```

This case can be easily solved by instantiating the existential quantifier and applying simplification.

4.3.2 Intermediate Analysis

An important observation from the previous attack is that the main attack analysis device of the inductive approach – the Dolev-Yao attacker `Spy` and the related infrastructure – play almost no role in it: possible injections of `Spy`-events into successful, i.e., finishing, traces of the cocaine protocol are merely those where the Spy feeds messages after Step 0 of the cocaine protocol. Even though `Spy` is able to play in `Fake` messages at any point of a partially finished trace this will lead to this trace ending unsuccessfully without reaching the goal of the auction. In the formal model, this is due to the fact that all latter steps require as a precondition that the previous message was one either originating from the `Server` or one from one of the `Friends j` for `j < friends`. Now, since `Spy, Server`, and `Friend j` are elements of the datatype `agent` that are created by different constructors, they are pairwise distinct. In particular, `Spy` cannot match either `Server` or `Friend j` for any `j` and the preconditions for any of the rules, `CAi` or `CAn` cannot become true any more once `Spy` has interspersed a trace by sending a fake message.

This limitation of the inductive approach is not surprising since modelling the agents as constructors of a datatype feeds into the global view (already discussed above) that agents are firmly divided into "bad" and "good".

Surprisingly, this does not impede the analysis of the sweetheart deal. On the contrary, for an analysis of insider threats in general, the fixed distinction of attackers and "good" principals is generally an inadequate modelling decision. As already observed in earlier papers on the formal analysis of insider threats [18], one of the major tricks to find attacks on security protocols is to consider insiders: the classic attack on the Needham-Schroeder attack is performed by the insider Eve[4]. This man-in-the-middle attack (or mirror-attack) uses impersonation which has motivated the Isabelle insider approach of using a sociological model inspired by Max Weber supported with Hempel and Oppenheim's logic of explanation to model and analyze insider threats in Isabelle with logic and proof.

Therefore, at this point we extend the inductive approach with the Isabelle insider framework that has been especially designed for the purpose. We only introduce the minimally necessary parts of that framework in order to illustrate how it can be used to model ringing. For more detail, the interested reader is referred to the main paper [2] and application examples to IoT insider threats [19] and insider threats to Airplane safety and security [20]. The following section recapitulates the parts of the Isabelle insider framework that are used to extend the inductive approach because they are needed to express collusion (see theorem `Insider_homo_oeconomicus` below).

[4]Eve uses her own legal credentials (a public key) to get Alice's nonce sent to Eve when Alice wants to communicate with her. This nonce is used as an authentication token to Bob. Eve next uses the first protocol run with Alice as an oracle to decrypt Bob's Nonce sent back to Eve encrypted with Alice's public key to challenge her presumed identity.

4.3.3 Isabelle Insider Framework

The Isabelle insider framework uses a taxonomy of insider threats [21]. This taxonomy is based on a thorough survey on results from counterproductive workplace behaviour, e.g., [22, 23] and case studies from the CMU-CERT Insider Threat Guide [8]. The insider framework simply models the taxonomy in HOL as datatypes, a concept of HOL that resembles the concept of taxonomy classes. As an example, consider the formal representation of *Psychological State* [21] as a datatype.

```
datatype psy_states = happy | depressed | disgruntled | angry | stressed
```

The element on the right hand side are the five injective constructors of the new datatype `psy_states`. They are simple constants, modelled as functions without arguments. Another example is *Motivation* [21].

```
datatype motivations = financial | political | revenge
                     | fun | competitive_advantage | power | peer_recognition
```

In the Isabelle insider framework, we combine the characteristics about the actor in a combined state.

```
datatype actor_state = State motivation psy_state
```

The *Precipitating Event* or *Catalyst* can be any event that has the potential to tip the insider over the edge into becoming a threat to their employer. It has been called the 'tipping point' in the literature. This catalyst is encoded as a tipping point predicate describing the psychological state and motivation of an actor to become an insider.

```
definition tipping_point :: actor_state ⇒ bool
 tipping_point a ≡ motivation a ≠ {} ∧ happy ≠ psy_states a
```

Insider threat case studies show that a recurring scheme in insider attacks lies in role identification as described in [18]. The Isabelle insider framework uses this role identification in the definition of the `UasI` predicate. It expresses that the insider plays a loyal member of an organization while he simultaneously acts as an attacker. Note, that in order to integrate the Isabelle insider framework with the inductive approach, we use the `agent` constructor `Friend` here.

```
 UasI a b ≡ (Friend a = Friend b)
```

Insider attacks link the insider characterization of psychological disposition with the above insider behaviour `UasI`. This is defined by the following rule `Insider a C` for the attacker `a`. The parameter `C` is a set of identities representing the members of an organization that are to be considered as safe.

```
Insider a C ≡ tipping_point (astate a) ⟶ (∀ b ∈ C. UasI a b)
```

Although the above insider predicate is a rule, it is not axiomatized. It is just an Isabelle definition, that is, it serves as an abbreviation. To use it in an application, like the auction protocol, we can use this rule as a local assumption in theorems (see below theorem `Insider_homo_oeconomicus`) or using the `assumes` feature of locales [24]).

4.3.4 Homo Oeconomicus and Ringing Attack

The principle of *homo oeconomicus* defines agents to be rational. In general, this economic principle captures the idea that any agent a will not spend more than necessary to get an asset, that is, if a can get the asset for price X, he will not pay price $Y > X$ to get it.

Without explicitly introducing the additional concept of "price" and buying assets, we can simply formalize the principle homo_oeconomicus for the context of the cocaine protocol, by stating that an agent that is currently the winner of round i will not make another bid in the next round. Technically, as a general property this states that for all traces t representing (intermediate) runs of the cocaine protocol no bidder will make a bid in the current round, if he is the highest bidder in the trace leading up to the current round. This is represented by the function CAtl applied to cocaine auction trace t. We define this function CAtl as a primitive recursive function that cuts off from any trace t all leading events if these exist at the front of t corresponding to (a) the Server's final broadcast according to rule CAn, (b) the last bid, i.e., the anonymous broadcast by some Friend j according to rule CAi, (c) all initial Server messages according to rule CA0 leaving the empty trace. Excluding that the currently highest bidder will make the next bid, corresponds to saying that the *head* of any trail t cannot be an event in which this bidder broadcasts his "yes". We can simply use the Isabelle list function hd since literally the first element of that list of "yes" broadcast events corresponding to rule CAi is the message to the Server. The auxiliary functions highest_bidder and cur_round are also defined as primitive recursive functions over traces in Isabelle in the intuitive way (for details see the Isabelle files [17]).

```
homo_oeconomicus ≡
∀ t ∈ cocaine_auction. ∀ j < friends.
  highest_bidder (CAtl t) (Friend j) ⟶
    hd t ≠ Says (Friend friends) Server
      (Crypt (pubK Server) {Key(pubK(Friend j)), Number(bid(cur_round t))})
```

As a first illustration for the use of this definition, we can use it to show that if there is only one bidder, the seller will only get the reserve price bid 1.

```
theorem homo_economicus_one_bidder:
friends = 1 ⟹ homo_oeconomicus
⟹ ∀ t ∈ cocaine_auction.
  t = [Says Server (Friend k) (Crypt(pubK(Friend j)){Number(bid i), Number msg}).
       k ← [0..<friends]] @ evs
⟶ i = 1
```

The above property is a useful stepping stone on the way to proving that if there is a collusion amongst all bidders, the Server will only get the reserve price. We cannot prove that a collusion between players glues them together to become physically one Friend corresponding to showing that the constant friends must be equal to 1. However, we can prove that the same conclusion as in the previous theorem follows as well. We assume that one bidder, Friend 0 is an insider and at the tipping point, and all bidders *act* as one agent, that is, the insider can impersonate them. From that we show the same conclusion as in the previous theorem follows: the seller only gets the reserve price.

```
theorem Insider_homo_oeconomicus:
homo_oeconomicus ⟹ tipping_point(astate 0) ⟹ Insider 0 {i. i < friends}
⟹ ∀ t ∈ cocaine_auction.
  t = [Says Server (Friend k) (Crypt (pubK(Friend j)) {Number(bid i), Number msg}).
       k ← [0..<friends]] @ evs
⟶ i = 1
```

In the proof of this theorem, the insider assumption is used to show that the prerequisite highest_bidder (CAtl t) (Friend j) of the hypothesis homo_oeconomicus can be made true for any bidder as soon as one of them, here Friend 0, is the highest bidder, because he can impersonate anyone. Invoking the assumption homo_oeconomicus, we can then prove that any continuation of the trace containing a first bid cannot continue, since it would be a bid of the same bidder contradicting the principle. So all traces

end with the highest bid `bid 1` which corresponds intuitively to a "reserve price" (`bid 0` is specified to be `0`, see Section 4.2).

We have thus formally shown that the insider assumption in fact enforces that the cocaine protocol can generally be corrupted by the collusion of all bidders.

5 Usefulness and Limitations

In the protocol we make several assumptions. We share the view of the authors of the original paper [4] "We do not believe that all such attacks can be detected, let alone stopped, by any particular auction protocol".

One problem is inherent in the set-up of the auction. In order to avoid that the other participants see who has made a bid at a particular time, the authors of [4] discuss that the participants have a clicker in their pockets and can press them without the others noticing. Then there are some problems with this, since once a bid has been made another press of a button would mean to bid for the next higher price. For example, assume that in each step the price goes up by 1000 and that the current high bid is at 49,000. Assume furthermore that both bidder b_1 and b_2 are willing to bid up to (inclusively) 50,000. They both decide to press, but b_1 is a split second faster and makes the bid for 50,000. The click by b_2 is still registered but would count as 51,000, an amount b_2 would not want to pay. Practically it would make sense that after a click any further clicks are disabled by a fixed amount of time (e.g., 10 seconds) and the amount of the current high bid is announced (e.g., on a display). In this scenario b_1's bid would initially disable the bidding process and after the display of 50,000 b_2 would have to press again before a further bid is registered. We assume that this process cannot be manipulated, since otherwise the auctioneer could always broadcast the acknowledgement of the bid with his sweetheart's key and all other bidders assume that they had been too slow to win the round, although actually one of them should have won the round and the sweetheart did actually not bid. The participants could not detect the manipulations since the actually generated trace is a legal trace of the protocol.

The proof guarantees only that the key used by the winner of the penultimate round – after the 30 seconds have lapsed without a bid – is used for the broadcast with the secret location, so that only the winner can decrypt the location. If, however, the true winner did not know that he had won the penultimate round, he would not expect to be sent the location and would not be in the position to detect foul play.

It is not surprising that without any assumptions only very little can be said about possible traces. In this case, it can be said that it can be detected if different keys are used in the penultimate and in the ultimate rounds. Obviously, the protocol cannot rule out that the seller sends a wrong *MeetingAppointment* to the true winner and uses communication channels outside the protocol to communicate the true *MeetingAppointment* to his sweetheart.

In summary, our model abstracts from some implementation details and inherently assumes the following:

- The implementation of the auction needs to provide a mechanism to avoid racing conditions and give unambiguous feedback to the successful bidder, for instance, some notice board and time delays between bids.
- The veracity of the meeting point is assumed and the post-procedure of the cocaine-money exchange is beyond the protocol model.

In fact, the part of the Isabelle insider framework that has not been used here provides the possibility to express infrastructures in which agents act and could thus be used to address the second point. However, that would be beyond the limits of this paper.

An interesting question is, how in the inductive approach, the attack on the Needham-Schroeder asymmetric protocol (NS-protocol) has been modelled. This man-in-the-middle attack can be considered as the first insider attack [18]. However, in the inductive approach the attacker is always only the agent `Spy`, and `Spy` is different from all other agents by construction. So, how could the NS-protocol be modelled when the attacker is `Friend i` for some `i` in `n`? The answer is that the protocol definition deliberately allows any agent in the rules for the inductive definition. In the rules of the definition of the NS-protocol in the inductive approach, letters `A` and `B` are used to suggest agents Alice and Bob. Consider, for example the crucial rule `NS1` from the NS-protocol formalization in the inductive approach where the initiator sends a nonce to the intended recipient.

```
NS1:  evs1 ∈ ns_public ⟹  Nonce NA ∉ used evs1 ⟹
       Says A B (Crypt (pubEK B) { Nonce NA, Agent A})
           # evs1  ∈  ns_public
```

The important point for the NS-protocol attack to work is that this intended recipient can be an attacker. That is, a legal participant of the network of peers is malicious and abuses a connection request by the initiator to impersonate this initiator. Although in the above rule, the capital letters `A` and `B` seem to indicate that these are the agents Alice and Bob, they are variables fixed only in the context of the rule. When the rule is applied they can be freely instantiated to any agent, also to `Spy`.

Summarizing, the inductive approach allows for different "kinds" of specification of a protocol: one where the actors are explicitly made distinct using different constructors of datatype agent in the specification (this is how we used it for the cocaine auction protocol) and one where agents are all abstract within the protocol (either as higher order variables as in the NS-application above or all represented as `Friend j`.) If we were to redefine the cocaine auction protocol in the latter way with abstract actors for all roles, then we would replace the seller also by `Friend k`. In this case, we would not be able to exclude the sweetheart deal: if we assume that the Server, say `Friend 0`, is an insider and at tipping point, he can impersonate a bidder and can then make his own bids using another role, say Friend j. This would enable the Server to provide a suitable bid for his sweetheart. In addition, it would enable another attack in which the Server, acting like a bidder, just drives the price up.

The formalization and proofs presented in this paper provide in summary the following results:

- Formal model of the cocaine auction protocol using the inductive approach proving the absence of sweetheart deals and the impossibility to exclude collusion.
- Formalization of arbitrary numbers of rounds, broadcast, and anonymous message sending for the inductive approach.
- The inductive approach can only deal with insider threats by abstracting from its agent datatype that prevents good agents from behaving badly.
- Integrating (parts of) the Isabelle insider framework with the inductive approach enables reasoning about collusion of insiders for auctions.
- The collusion exhibits that the assumption *homo oeconomicus* suffices to prove that rational insiders may use collusion to force the reserve price.

6 Moving to Reality

An approach based on formalized reasoning and code extraction allows for the generation of systems that brings the assurance that they have the required properties to an unprecedented level. The fact that

any system has limitations and that there may be ways to work around it does not mean that there is no benefit. In this section we want to discuss aspects of this and argue that the threshold for workarounds are so high that they are practically not viable.

Assumed we wanted to realize the approach practically. In this case, the auction would run on a *new* computer and the different systems for running the cocaine auction are installed in the presence of experts in the employment of the different parties concerned. For instance, an operating system is installed that is digitally signed, likewise a Scala environment, and the most recent Isabelle system. The parties convince themselves that the specification in Isabelle corresponds to a high-level description of what they expect. Then in the presence of the experts Isabelle is used to extract the code on the new computer, the code is compiled, and then used to run the auction.

6.1 Overview of Practical Solution

For a practical solution, the inductive definition of the cocaine auction protocol represents the specification of a practically applicable protocol. Following general software engineering principles, a formal specification is implemented correctly if a program in an executable programming language, like Scala, meets the specification – we say the implementation *conforms* to the specification. We apply this principle using Isabelle in a completely rigorous way by executing the following steps:

- We provide an Isabelle definition of a constructive test predicate for correct cocaine auction traces.
- We generate executable code for this test predicate in the programming language Scala using Isabelle's code generation mechanism [25]. Note that Scala is a functional programming language which runs on the Java virtual machine. As such Scala is a language that is particularly interesting for industrial applications, since it shares the property of Java that it is platform independent; furthermore it is compatible with Java in that it is possible to integrate Scala programs with Java programs. The extraction mechanism in Isabelle allows to transform computational definitions in Isabelle to functions in Scala. As a consequence, the Scala code shares its properties with the corresponding definitions in Isabelle and these properties are formally guaranteed by the proofs in the Isabelle system.
- We prove correctness of the test predicate: we additionally provide proofs that a trace of events that is accepted by the test function is also in the set of all cocaine auction traces defined in its specification in Section 4.2. This proves that our test predicate identifies only correct traces of cocaine auctions.

We next give the specification of the test predicate in Isabelle and show how Isabelle's code generation features are evoked. The generated Scala code is contained in an Appendix. Finally, we introduce the proved correctness properties, explaining their significance and the abstract proof ideas but omitting the Isabelle proofs scripts. The entire development including the code generation definition and correctness proofs is available online [17].

6.2 Definition of Test Predicate CA_n and Code Generation

The inductive definition of the cocaine auction protocol in Section 4.2 specifies a fairly concrete predicate[5] which identifies all traces of events that constitute possible executions of the protocol. However, this predicate is in itself not executable nor can Isabelle generate code from it. One technical impediment for generating code is that any trace of a cocaine auction uses the `pubK` function of the inductive

[5] Any set in HOL is equivalent to its defining predicate.

package. This function assigns public keys to agents and is formalized abstractly as a constant with no definition. Isabelle cannot generate code from an abstract definition. An executable test predicate must be able to check whether the natural numbers contained at the corresponding position within the events representing a round `bid i` of bidding in a trace is equal to the public key `pubK(Friend j)` of the bidder `Friend j`. This check could be achieved by assuming that the inverse of the `pubK` function is known. The inverse of the public key is the private key so this is deemed to be insecure. Instead, we may safely assume for the practical realization of a test predicate that all bidders enter their *public* keys anonymously to a list of *known keys* `kl`. The constructive function `CA_i'''` below checks in each round `i` whether the public key used is contained in this list of all public keys of bidders. For reasons of clarity of the proof structure, we use an additional intermediate definition of the check predicate for each round – called `CA_i` – that uses an existential quantifier over `j` to allow referring to `Friend j`.

In summary, the definition of the test predicate `CA_n` uses a refinement from non-constructive function `CA_i` over to the constructive `CA_i'''`. The conformance of `CA_i'''` to `CA_i` and in turn to the specification `cocaine_auction` is subject of the correctness proofs in the subsequent section.

We present the definition of the test predicates top down starting from the global test predicate `CA_n` and its constructive refinement `CA_n'''`. Both functions return true if their first argument is a trace `t` that represents a "transcript" of a correct cocaine auction. In fact, the toplevel definitions of the test functions `CA_n` and `CA_n'''` only differ in that the former uses `CA_i`.

```
definition CA_n :: [event list, nat, nat, nat, nat ⇒ nat, nat, nat] ⇒ bool
where CA_n t pF f pS b n mtng ≡ (CA_fn (take f t) pF f (b n) mtng) ∧ 0 < n ∧
                                (CA_i (drop f t) pF f b pS n)
```

and the latter uses `CA_i'''` instead.

```
definition CA_n'''':: [event list, nat, nat, nat, nat ⇒ nat, nat, nat, nat list] ⇒ bool
where CA_n''' t pF f pS b n mtng kl ≡ (CA_fn (take f t) pF f (b n) mtng) ∧ 0 < n ∧
                                (CA_i''' (drop f t) Pf f b pS n kl)
```

Both of these test predicates use the sub-predicate `CA_fn` that implements the check of the *n*th step `CAn` of the protocol by comparing to a list of events that represents the last broadcast message of the Server announcing the winner and sending the meeting place confidentially.

```
definition CA_fn :: [event list, nat, nat, nat, nat] ⇒ bool
where CA_fn t pF f bi mtng ≡
  (t = [Says Server (Friend i) (Crypt pF { Number bi, Number mtng }). i ← [0..<f]])
```

As seen in its application within `CA_n` and `CA_n'''` the test function `CA_fn` is meaningfully applied only to an initial segment of length `friends` of an event trace `t` (which is cut off using the list operation `take`). The remainder of any trace `t` after cutting off `f` elements is produced by the complementary expression `drop f t`. It allows invocation of the test predicates for the `i` rounds on the rest of a trace conjoining the resulting predicate test results of type bool by the logical "and" operator ∧.

The non-constructive version `CA_i` is represented as the following primitive recursive function definition. In summary, its role is to check in each round `i` whether the initial trace segment of length `Suc friends` is a bid by one of the bidders represented by the public key of a friend. This checking of a round is given by the sub-predicate `CA_fi` (see below) in `CA_i_step_n` and continues recursively until the round counter reaches `0`. This final case is defined by `CA_i_step_0` which maps to the constructive predicate `CA_f0` checking the base step (see below).

```
primrec CA_i :: [event list, nat, nat, nat ⇒ nat, nat, nat] ⇒ bool
where
```

```
CA_i_step_0: CA_i l pF f b pS 0 = CA_f0 l pS f |
CA_i_step_n: CA_i l pF f b pS (Suc i) =
               ((CA_fi (take (Suc f) l) pF f (b (Suc i))  pS) ∧
               (∃ j. j < friends ∧ CA_i (drop (Suc f) l) (pubK (Friend j)) f b pS i))
```

The predicate CA_f0 signifies that a sequence of events corresponds to the initial segment of any cocaine auction in which the Server sends out his key to all bidders as specified in the rule CA0 in Section 4.2.

```
definition CA_f0 :: [event list, nat, nat] ⇒ bool
where CA_f0 t pS f ≡ (t = [Says Server (Friend i) (Key(pS)). i ← [0..<f]])
```

The predicate CA_fi represents a check that in each round i during the cocaine auction its prefix starts with a sequence of messages from one of the bidders to the server and all other bidders containing the bidder's public key and the current bid.

```
definition CA_fi :: [event list, nat, nat, nat, nat] ⇒ bool
where CA_fi t pF f bi pS ≡
 (t = Says (Friend f) Server
      (Crypt pS {Key pF , Number bi}) #
         [Says (Friend f) (Friend i) (Crypt pS {Key pF, Number bi}). i ← [0..<f]])
```

Note, that all the above definitions apart from CA_i are constructive, i.e., Scala code can be generated from them. The function CA_i uses the function pubK and is thus not constructive. Moreover, CA_i_step_n uses an existential quantifier to identify the public key coming next in the trace. Since public keys are natural numbers, this existential quantifier ranges over a potentially infinite domain. For both reasons, code cannot be generated from this definition. Fortunately, as discussed initially, we can refine this test predicate into a constructive one CA_i''' which is used in the constructive global test function CA_n''' above.

```
primrec CA_i'''':: [event list, nat, nat, nat ⇒ nat, nat, nat, nat list] ⇒ bool
where
CA_i'''_step_0: CA_i''' l pF f b pS 0 kl = CA_f0 l pS f |
CA_i'''_step_n: CA_i''' l pF f b pS (Suc i) kl =
                  ((CA_fi (take (Suc f) l) pF f (b (Suc i))  pS) ∧
                    (case (winner_i (drop (Suc f) l)) of
                          Some K => List.member kl K ∧
                                    (CA_i''' (drop (Suc f) l) K f b pS i kl)
                    | _ ⇒ False))
```

As the non-constructive predicate CA_i, the above recursive definition uses the same predicate CA_fn in each round to assess that the first elements correspond to the specification of step CAi of the protocol. However, the existential part of the recursion is replaced by a constructive choice that uses a function winner_i that computes the winner of the current trace, i.e., the public key that is in the top segment of the current trace. In addition, it checks that this winner is represented by a public key from the list kl of all public keys of all friends. The definition of the winner function is a "partial" function[6]: only if the event list given as input to winner_i contains an event of the form Says Server (Crypt SK {Key(K), N}) for suitable values of SK, K, N, it returns the key K.

```
primrec winner_i :: event list ⇒ key option
where  undef: winner_i [] = None
     | step : winner_i (X # evs) =
              (case X of (Says A Server M) ⇒
```

[6]All function in HOL are total. Nevertheless, the option type used here allows explicit modelling of partial functions.

```
        (case M of (Crypt SK { Key(K), N }) ⇒ Some K
        | _ ⇒ None)
| _ ⇒ None)
```

Note, how the additional use of the parameter `kl` allows to get rid of the existential quantifier making the predicate `CA_i'''` constructive but necessitating the additional assumption that the parameter `kl` is the list of all public keys of all bidders. Since `CA_i'''` is constructive we can now generate code from it and also from the constructive global predicate `CA_n'''`. The following single line urges Isabelle to generate Scala code for the list of functions provided.

export_code `CA_f0 CA_fi CA_fn  CA_i''' CA_n'''` **in** `Scala`

This automatically creates executable Scala code also for all library definitions used for cryptographic protocols. The code is fully contained in the Appendix.

6.3 Correctness of Test Predicate

The thorough implementation of the Isabelle framework guarantees us that the code that is automatically generated from the definitions presented in the previous section correctly implements these definitions. Apart from the informal argument presented in the previous section, we do not know whether the predicates `CA_n` and `CA_n'''` correctly implement the cocaine auction protocol. By correct implementation, or short "correctness" of the test functions, we mean that any list of events that is positively identified by the test predicates is in fact a trace of a cocaine auction in the sense of its definition by the inductive definition `cocaine_auction` in Section 4.2. Formally, we want to prove for all traces of events `evs` the following *correctness* property.

```
CA_n''' evs ... ⟹ evs ∈ cocaine_auction
```

It now also becomes clear why we kept the definition of the non-constructive predicate `CA_n'''` as an intermediate refinement step: practically, we prove correctness for the definition of `CA_f0` and `CA_i` which quite straightforwardly implies the same property for `CA_n`. We then prove that `CA_i''' evs` implies `CA_i evs` for all rounds `i` which in turn allows us to immediately infer correctness for `CA_n'''` but the proof for `CA_i` is much simpler and more elegant.

More precisely, we first prove the following lemma for the test predicate `CA_f0` that identifies traces according to step `CA0` of the cocaine auction.

lemma `CA_f0_CA0: CA_f0 t (pubK Server) friends ⟹ t ∈ cocaine_auction`

The proof is a simple unfolding of the definition of `CA_f0` followed by applying rule `CA0`.

Next we prove that if the predicate `CA_i` returns true for a trace `evs` this implies that the trace has been produced by an arbitrary number of rounds according to rule `CAi` and is thus a cocaine auction trace.

```
lemma forall_CA_i_imp_CA:
  ∀ i. ∀ j. ∀ evs.
     j < friends ∧ CA_i evs (pubK (Friend j)) friends bid (pubK Server) i
     ⟶ evs ∈ cocaine_auction
```

The proof is a natural number induction over the number of rounds `i`. Correctness of the test function `CA_i` needs now only to be combined with that of `CA_fn` testing the last step of the cocaine auction protocol as expressed in the following lemma.

```
lemma CA_fn_CAn:
  CA_fn t (pubK (Friend j)) friends (bid i) mtng ⟹ j < friends ⟹ 0 < i ⟹
  evs = t @ evsf; CA_i evsf (pubK (Friend j)) friends bid (pubK Server) i ⟹
  evs ∈ cocaine_auction
```

From there, we can derive rather straightforwardly the correctness property for the non-constructive predicate CA_n.

```
theorem CAn_CA: CA_n evs (pubK (Friend j)) friends (pubK Server) bid i mtng ⟹
   j < friends ⟹ evs ∈ cocaine_auction
```

After unfolding the definition of CA_n, the proof essentially applies the previous lemma.

Finally, it remains to show that CA_i''' implies CA_i. To this end, we need to construct a proof by induction. Within the induction the invocation of the induction hypothesis necessitates the following lemma showing that if CA_i''' holds for round Suc i then also for round i. The first assumption defines the set kl as the set of all public keys of all bidders: our main assumption for constructiveness.

```
lemma CA_i'''_Suc_i_i: kl = [(pubK (Friend k)). k ← [0..<friends]] ⟹
       j < friends ⟹ CA_i''' l (pubK (Friend j)) f b pS (Suc i) kl ⟹
       ∃ ja < friends. CA_i''' (drop (Suc f) l) (pubK (Friend ja)) f b pS i kl
```

Applying induction for natural numbers again over i, the number of rounds allows now a straightforward proof of the key lemma.

```
lemma CA_i'''_CA_i: kl = [(pubK (Friend k)). k ← [0..<friends]] ⟹
             ∀ j < friends. ∀ l. CA_i''' l (pubK (Friend j)) f b pS i kl ⟹
                     CA_i l (pubK (Friend j)) f b pS i
```

This now allows reducing the correctness for the constructive test predicate CA_n''' to that of the non-constructive intermediate function CA_n. Thus, we can prove correctness of the cocaine auction test predicate CA_n''' from which the Scala code in the Appendix is generated automatically by Isabelle.

```
theorem CAn'''_CA: kl = [(pubK (Friend k)). k ← [0..<friends]] ⟹
    CA_n''' evs (pubK (Friend j)) friends (pubK Server) bid i mtng kl ⟹
    j < friends ⟹ evs ∈ cocaine_auction
```

6.4 Discussion of Practical Solution

In the following we look at potential weaknesses of the approach and conclude that although each system has limitations, workarounds are now so difficult and expensive that they are not practically viable.

If we are extremely sceptical then even for such a scenario attacks cannot be completely ruled out but may occur on different levels:

- The computer hardware has been tempered with and whatever kind of operating system we try to install, the system just pretends to install the system, and whatever kind of program we compile another program will actually run.
- The Scala compiler has been changed so that it does not actually compile the Scala code in a way that we expect.
- The Isabelle system has been changed so that the extracted code is not accurate.

Let us assume that each of these steps produces something that is close enough to what we all expect but changes the behaviour substantially without the experts noticing. This requires substantial amounts of efforts, designing a chip, getting hold of the production/delivery line of the computer, building a new compiler and penetrating into a certification agency to falsify a certificate, hijacking the Isabelle system. All this is *thinkable*, but will typically be very difficult to achieve and very expensive. To look at a historic precedent for instance, in case of the Stuxnet worm [26] it is assumed that the USA and Israel were behind the attack and that it required a substantial amount of work. Still most of these attacks can be made even more difficult in that each expert brings their own trusted system and all these are run in parallel and matters proceed only if all systems agree.

The situation is in some way analogous to reinforcing a house against burglaries. There is a big difference between leaving the door wide open and having a special door and other reinforcements that can be penetrated only under great difficulty, causing a lot of work and taking considerable time. From a certain point on, the work and the increased risk make a burglary look unattractive and poor value for money. Likewise in the case of the cocaine auction, the effort to practically cheat the system looks very high compared to the potential gain.

We should also note that practically we assume that all the critical software is written in Isabelle and their formal properties are proved in Isabelle. The properties need human inspection in order to check that they actually are the desired properties. The proofs, however, need no human inspection. The Isabelle system guarantees their correctness and the proofs are checked by Isabelle after starting the system before the code is extracted. This way, the most significant amount of work is done by the Isabelle system (of course only after the proofs were interactively generated, in this case by the first author of this paper).

A practical system would contain other non-critical parts such as its graphical user interface which are not generated via Isabelle. These parts could typically be kept small and simple and would need thorough inspection by the experts of the parties involved.

7 Conclusions

In this paper, we have investigated the vulnerability of auctions to insider attacks in particular different forms of collusion. We used the cocaine auction protocol as a case study that takes mistrust to an extreme. Using modelling and analysis in Isabelle we experimented with two different approaches, the inductive approach to security protocols and the Isabelle insider framework. We were able to model the protocol in the inductive approach and show that its formal specification excludes a possible insider attack, the "sweetheart deal". Integrating the inductive approach with the Isabelle insider framework enabled showing that collusion between all bidders, so-called "ringing", cannot be excluded. In order to prove the latter theorem, we formalized a notion of "homo-oeconomicus" for the cocaine auction protocol.

In addition to the earlier paper [5], we provided a practical implementation of a constructive test predicate that realizes the protocol in Scala. The code has been generated by a mechanism of Isabelle to generate Scala code from constructive definitions. Moreover, we proved correctness of the constructive test predicate in Isabelle with respect to the inductive definition of the protocol.

Isabelle is a very general and very powerful tool that has been successfully applied in a broad range of applications. The application areas include advanced pure mathematics (for instance, some properties relevant in the proof of the Kepler conjecture [27]), economic applications such as proofs of Arrow's theorem [28], properties of algorithms (such as Dijkstra's shortest path algorithm [29]) and many more areas as found in the Archive of Formal Proofs[7]. More specifically relevant to the current work are proofs

[7] See, `https://www.isa-afp.org/`.

on auctions [1], proofs on verifying cryptographic protocols [30], and proofs on Insider threats [2]. The current work reuses those previous works integrating them to enable the formalisation of an auction protocol that uses broadcasting to enable anonymity. While some of the issues around broadcasting have been addressed in more recent extensions of the inductive approach [31] for group protocols, we provide our own solutions tailored to the specific needs of insider threats.

An important alternative approach to our approach to machine supported verification of security protocols is the use of specialised model checkers, for example, Avispa [32], Proverif [33], or the earlier Casper [15]. The main advantage of model checking is the full automation of the verification process: model checking is a "push-button" technique of analysis. The immediate disadvantage of this approach is the limitation to finite models – and the accompanying "state explosion problem" – which requires strong abstraction when modelling security protocols. From a protocol modelling and analysis point of view, we believe that the bigger problem with model checking is that predefined implemented abstractions need to be used when modelling protocols. For example, `witness` or `authentication_on` are fixed predicates that have the quality of keywords in Avispa allowing the specification of assertions within protocols and in properties to be verified but their semantics is practically in the eye of the beholder: there is a clear semantics but does it match the intuition of the user who defines the specification? And is it also clear to the user who reads the specification and needs to take these keywords as guarantees of security properties?

The Isabelle approach, by comparison, comes with more effort on the side of proofs of security properties but the expressive and flexible modelling capabilities of Higher Order Logic enable a better understanding of models since the basis is common logical and mathematical language.

Limitations of our model are discussed in the previous section showing that – despite the formality introduced and frameworks used – all guarantees depend on the abstraction we chose when modelling. Any formalization and proof of system properties depends always on the model we consider. This is also true for the system abstractions of protocols and auctions. Therefore, the implicit assumptions about real world participants are crucial. The use of the insider framework makes role impersonation explicit in models and therefore helps to understand in more detail how insider attacks work in auctions. The additional assumption *homo oeconomicus* could be a beneficial extension to enrich the Isabelle insider framework by a notion of a rational insider although its general assumption as part of an Insider definition is disputable.

Future research includes exploring whether the additional assumptions we encountered here are more generally valid for comparable insider attack scenarios. We are currently interested in applying the Isabelle insider framework to IoT system scenarios in particular for cost effective health care systems, for example, using smart phones and other smart devices for monitoring for the diagnosis of Alzheimer's disease [34]. In this context, insider threats at the organisational level are challenging but also protocols that are in use to communicate sensitive data to servers for different purposes: anonymised data collected for research purposes, complete patient data to hospital servers for monitoring and diagnosis, but also partially sanitized data to health insurers. Another avenue for research could be to generalise the combination of security protocol formalisation and code generation in Isabelle. The current additional move to reality appears quite successful. So it seems promising to extract the tricks applied for refining the abstract inductive protocol specification to constructive test predicates. Those mechanisms might constitute a generic mechanism for producing code from (security) protocol specifications.

Acknowledgement

Part of the research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318003 (TRE$_S$PASS). This publi-

cation reflects only the authors' views and the Union is not liable for any use that may be made of the information contained herein.

References

[1] M. B. Caminati, M. Kerber, C. Lange, and C. Rowat, "Sound auction specification and implementation," in *Proc. of the 16th ACM Conference on Economics and Computation (EC'15), Portland, Oregon, USA*. ACM, June 2015, pp. 547–564.

[2] F. Kammüller and C. W. Probst, "Modeling and verification of insider threats using logical analysis," *IEEE Systems Journal*, pp. 1–12, August 2015.

[3] L. C. Paulson, "The inductive approach to verifying cryptographic protocols," *Journal of Computer Security*, vol. 6, no. 1-2, pp. 85–128, September 1998.

[4] F. Stajano and R. Anderson, "The cocaine auction protocol: On the power of anonymous broadcast," in *Proc. of the 3rd International Workshop Information Hiding (IH'99), Dresden, Germany*, ser. Lecture Notes in Computer Science, vol. 1768. Springer, Berlin, Heidelberg, September 1999, pp. 434–447.

[5] F. Kammüller, M. Kerber, and C. Probst, "Towards formal analysis of insider threats for auctions," in *Proc. of the 8th ACM CCS International Workshop on Managing Insider Security Threats (MIST'16), Vienna, Austria*. ACM, October 2016, pp. 23–34.

[6] P. Klemperer, *Auctions: Theory and Practice*. Princeton University Press, 2004.

[7] V. Krishna, *Auction Theory*. Academic Press, 2002.

[8] D. M. Cappelli, A. P. Moore, and R. F. Trzeciak, *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*. Addison-Wesley Professional, February 2012.

[9] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of Cryptology*, vol. 1, no. 1, pp. 65–75, January 1988.

[10] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions in Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.

[11] U. Maurer and S. Wolf, "The Diffie-Hellman protocol," *Designs, Codes and Cryptography*, vol. 19, no. 2, pp. 147–171, March 2000.

[12] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

[13] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proc. of the 22nd ACM Conference on Computer and Communications Security (CCS'15), Denver, Colorado, USA*. ACM, October 2015, pp. 5–17.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel, "Isabelle/HOL - a proof assistant for higher-order logic," ser. Lecture Notes in Computer Science, vol. 2283. Springer-Verlag Berlin Heidelberg, 2002.

[15] G. Lowe, "Casper: A compiler for the analysis of security protocols," in *Proc. of the 10th Computer Security Foundations Workshop (CSFW '97), Rockport, Massachusetts, USA*, June 1997, pp. 18–30.

[16] D. Dolev and A. C. Yao, "On the security of public key protocols," in *Proc. of the 22nd Annual Symposium on Foundations of Computer Science (SFCS'81), Nashville, Tennessee, USA*. IEEE, October 1981, pp. 350–357.

[17] F. Kammüller, "Isabelle formalisation of cocaine auction protocol," https://www.dropbox.com/sh/uyku2q2ofb69cwm/AACYcxAnuI75YIOyqdz4wcoua?dl=0 [Online; Accessed on March 1, 2017].

[18] F. Kammüller and C. W. Probst, "Combining generated data models with formal invalidation for insider threat analysis," in *Proc. of the 2014 IEEE Security and Privacy Workshops (SPW'14), San Jose, California, USA*. IEEE, May 2014, pp. 229–235.

[19] F. Kammüller, J. R. C. Nurse, and C. W. Probst, "Attack tree analysis for insider threats on the IoT using Isabelle," in *Proc. of the 4th International Conference on Human Aspects of Information Security, Privacy,*

and Trust (HAS'15), Toronto, Ontario, Canada, ser. Lecture Notes in Computer Science, vol. 9750. Springer, Cham, July 2016, pp. 234–246.

[20] F. Kammüller and M. Kerber, "Investigating airplane safety and security against insider threats using logical modeling," in *Proc. of the 2016 IEEE Security and Privacy Workshops (SPW'16), San Jose, California, USA*. IEEE, May 2016, pp. 304–313.

[21] J. R. C. Nurse, O. Buckley, P. A. Legg, M. Goldsmith, S. Creese, G. R. T. Wright, and M. Whitty, "Understanding insider threat: A framework for characterising attacks," in *Proc. of the 2014 IEEE Security and Privacy Workshops (SPW'14), San Jose, California, USA*. IEEE, May 2014, pp. 214–228.

[22] M. J. Martinko, M. J. Grundlach, and S. C. Douglas, "Toward an integrative theory of counterproductive workplace behaviour," *International Journal of Selection and Assessment*, vol. 10, no. 1–2, pp. 36–50, January 2003.

[23] B. Marcu and H. Schuler, "Antecedents of counterproductive behaviour at work: a general perspective," *Journal of Applied Psychology*, vol. 89, no. 4, pp. 647–660, September 2004.

[24] F. Kammüller, M. Wenzel, and L. C. Paulson, "Locales a sectioning concept for Isabelle," in *Proc. of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99), Nice, France*, ser. Lecture Note in Computer Science, vol. 1690. Springer, Berlin, Heidelberg, September 1999, pp. 149–165.

[25] F. Haftmann, "Code generation from Isabelle/HOL theories," www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/codegen.pdf [Online; Accessed on March 1, 2017].

[26] D. Kushner, "The real story of Stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48–53, March 2013.

[27] G. Bauer and T. Nipkow, "Flyspeck I: Tame Graphs," https://www.isa-afp.org/entries/Flyspeck-Tame.shtml [Online; Accessed on March 1, 2017].

[28] T. Nipkow, "Social choice theory in HOL: Arrow and Gibbard-Satterthwaite," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 289–304, October 2009.

[29] B. Nordhoff and P. Lammich, "Dijkstra's shortest path algorithm," https://www.isa-afp.org/entries/Dijkstra_Shortest_Path.shtml [Online; Accessed on March 1, 2017].

[30] L. C. Paulson, "Proving properties of security protocols by induction," in *Proc. of the 10th Computer Security Foundation Workshop (CSFW '97), Rockport, Massachusetts, USA*. IEEE, June 1997, pp. 70–83.

[31] J. E. Martina and L. C. Paulson, "Verifying multicast-based security protocols using the inductive method," *International Journal of Information Security*, vol. 14, no. 2, pp. 187–204, 2015.

[32] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The avispa tool for the automated validation of internet security protocols and applications," in *Proc. of the 17th International Conference on Computer Aided Verification (CAV'05), Edinburgh, Scotland, United Kingdom*, ser. Lecture Notes in Computer Science, vol. 3576. Springer, Berlin, Heidelberg, July 2005, pp. 281–285.

[33] B. Blanchet, "Automatic verification of security protocols in the symbolic model: the verifier ProVerif," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. Lecture Notes in Computer Science, vol. 8604. Springer International Publishing, 2014, pp. 54–87.

[34] CHIST-ERA, "Success: Secure accessibility for the internet of things," http://www.chistera.eu/projects/success [Online; Accessed on March 1, 2017].

Author Biography

Florian Kammüller is a Senior Lecturer in the Department Computer Science at Middlesex University London. He is the Programme Leader for the BSc in Computer Science and the MSc by Research in Security. His research focuses on Formal Methods, Security, and Distributed Systems. He is currently leading the CHIST-ERA project SUCCESS on Security on Privacy of the IoT.

Manfred Kerberis a Senior Lecturer in the School of Computer Science at the University of Birmingham. Currently his main research interests are applying formal approaches to economic problems, in particular in the area of auctions.

Christian W. Probst is an Associate Professor and head of the section for Cyber Security in the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, and director of studies of a B.Eng. study line Software Technology. His current research focuses on organizational security as well as embedded systems and compilers.

A Generated Scala Code

This appendix contains the Scale code automatically generated from the definitions in Section 6.2. Rather than inlining comments into the code, we document it altogether here in the beginning. Most of the code is generated from the underlying theories on cryptography and protocol primitives: objects `HOL`, `Nat`, `Message`, `Event` contain class definitions for the datatypes in those theories and method (or function) definitions for the functions defined in the respective Isabelle theories over those datatypes. Interesting parts of the generation process can be observed in the definitions of functions, for example, `equal_agent` and `equal_msg` that are due to the semantics of datatype definitions in Isabelle. Constructors of dataypes in Isabelle are injective functions and they are distinct. For example, if `Number(x2) = Number(y2)`, then `x2 = y2` because constructor `Number` is injective. Similarly, `Spy = Friend j => false`, that is, they are unequal for all `j` since both are constructors of dataype `agent`, hence distinct. Since these properties are not generally the case in Scala, those implicit properties of datatypes lead to a list of properties spelling out the corresponding formulas for all combinations of data type constructors.

Object `Lista` contains the portion of list functionality of Isabelle that we use in our specification of the cocaine auction protocol. For example, we compare lists using the generic Isabelle equality "=" but

since this is applied to a list type in our test predicate the code generator needs to create the corresponding "higher order" predicate `equal_list` in Scala.

Finally, the object `CocaineAuction` contains the code of the test predicates for which we requested code generation by adding the line

export_code `CA_f0 CA_fi CA_fn  CA_i''' CA_n'''` **in** `Scala`

to our Isabelle theory (see Section 6.2). For each of the listed test predicates, Isabelle generates the correspondingly named Scala functions. They correspond very closely to their Isabelle counterparts now that the underlying "machinery" has been provided. Note that the quotes behind the predicate name `CA_n'''` are omitted in the generation.

```
object HOL {

trait equal[A] {
  val ‘HOL.equal‘: (A, A) => Boolean
}
def equal[A](a: A, b: A)(implicit A: equal[A]): Boolean = A.‘HOL.equal‘(a, b)

def eq[A : equal](a: A, b: A): Boolean = equal[A](a, b)

} /* object HOL */

object Nat {

abstract sealed class nat
final case class zero_nat() extends nat
final case class Suc(a: nat) extends nat

def equal_nata(x0: nat, x1: nat): Boolean = (x0, x1) match {
  case (zero_nat(), Suc(x2)) => false
  case (Suc(x2), zero_nat()) => false
  case (Suc(x2), Suc(y2)) => equal_nata(x2, y2)
  case (zero_nat(), zero_nat()) => true
}

implicit def equal_nat: HOL.equal[nat] = new HOL.equal[nat] {
  val ‘HOL.equal‘ = (a: nat, b: nat) => equal_nata(a, b)
}

def less_eq_nat(x0: nat, n: nat): Boolean = (x0, n) match {
  case (Suc(m), n) => less_nat(m, n)
  case (zero_nat(), n) => true
}

def less_nat(m: nat, x1: nat): Boolean = (m, x1) match {
  case (m, Suc(n)) => less_eq_nat(m, n)
  case (n, zero_nat()) => false
}

} /* object Nat */

object Message {
```

```
abstract sealed class agent
final case class Server() extends agent
final case class Friend(a: Nat.nat) extends agent
final case class Spy() extends agent

abstract sealed class msg
final case class Agent(a: agent) extends msg
final case class Number(a: Nat.nat) extends msg
final case class Nonce(a: Nat.nat) extends msg
final case class Key(a: Nat.nat) extends msg
final case class Hash(a: msg) extends msg
final case class MPair(a: msg, b: msg) extends msg
final case class Crypt(a: Nat.nat, b: msg) extends msg

def equal_agent(x0: agent, x1: agent): Boolean = (x0, x1) match {
  case (Friend(x2), Spy()) => false
  case (Spy(), Friend(x2)) => false
  case (Server(), Spy()) => false
  case (Spy(), Server()) => false
  case (Server(), Friend(x2)) => false
  case (Friend(x2), Server()) => false
  case (Friend(x2), Friend(y2)) => Nat.equal_nata(x2, y2)
  case (Spy(), Spy()) => true
  case (Server(), Server()) => true
}

def equal_msg(x0: msg, x1: msg): Boolean = (x0, x1) match {
  case (MPair(x61, x62), Crypt(x71, x72)) => false
  case (Crypt(x71, x72), MPair(x61, x62)) => false
  case (Hash(x5), Crypt(x71, x72)) => false
  case (Crypt(x71, x72), Hash(x5)) => false
  case (Hash(x5), MPair(x61, x62)) => false
  case (MPair(x61, x62), Hash(x5)) => false
  case (Key(x4), Crypt(x71, x72)) => false
  case (Crypt(x71, x72), Key(x4)) => false
  case (Key(x4), MPair(x61, x62)) => false
  case (MPair(x61, x62), Key(x4)) => false
  case (Key(x4), Hash(x5)) => false
  case (Hash(x5), Key(x4)) => false
  case (Nonce(x3), Crypt(x71, x72)) => false
  case (Crypt(x71, x72), Nonce(x3)) => false
  case (Nonce(x3), MPair(x61, x62)) => false
  case (MPair(x61, x62), Nonce(x3)) => false
  case (Nonce(x3), Hash(x5)) => false
  case (Hash(x5), Nonce(x3)) => false
  case (Nonce(x3), Key(x4)) => false
  case (Key(x4), Nonce(x3)) => false
  case (Number(x2), Crypt(x71, x72)) => false
  case (Crypt(x71, x72), Number(x2)) => false
  case (Number(x2), MPair(x61, x62)) => false
  case (MPair(x61, x62), Number(x2)) => false
  case (Number(x2), Hash(x5)) => false
  case (Hash(x5), Number(x2)) => false
  case (Number(x2), Key(x4)) => false
```

```
  case (Key(x4), Number(x2)) => false
  case (Number(x2), Nonce(x3)) => false
  case (Nonce(x3), Number(x2)) => false
  case (Agent(x1), Crypt(x71, x72)) => false
  case (Crypt(x71, x72), Agent(x1)) => false
  case (Agent(x1), MPair(x61, x62)) => false
  case (MPair(x61, x62), Agent(x1)) => false
  case (Agent(x1), Hash(x5)) => false
  case (Hash(x5), Agent(x1)) => false
  case (Agent(x1), Key(x4)) => false
  case (Key(x4), Agent(x1)) => false
  case (Agent(x1), Nonce(x3)) => false
  case (Nonce(x3), Agent(x1)) => false
  case (Agent(x1), Number(x2)) => false
  case (Number(x2), Agent(x1)) => false
  case (Crypt(x71, x72), Crypt(y71, y72)) =>
    Nat.equal_nata(x71, y71) && equal_msg(x72, y72)
  case (MPair(x61, x62), MPair(y61, y62)) =>
    equal_msg(x61, y61) && equal_msg(x62, y62)
  case (Hash(x5), Hash(y5)) => equal_msg(x5, y5)
  case (Key(x4), Key(y4)) => Nat.equal_nata(x4, y4)
  case (Nonce(x3), Nonce(y3)) => Nat.equal_nata(x3, y3)
  case (Number(x2), Number(y2)) => Nat.equal_nata(x2, y2)
  case (Agent(x1), Agent(y1)) => equal_agent(x1, y1)
}

} /* object Message */

object Event {

abstract sealed class event
final case class Says(a: Message.agent, b: Message.agent, c: Message.msg)
  extends event
final case class Gets(a: Message.agent, b: Message.msg) extends event
final case class Notes(a: Message.agent, b: Message.msg) extends event

def equal_eventa(x0: event, x1: event): Boolean = (x0, x1) match {
  case (Gets(x21, x22), Notes(x31, x32)) => false
  case (Notes(x31, x32), Gets(x21, x22)) => false
  case (Says(x11, x12, x13), Notes(x31, x32)) => false
  case (Notes(x31, x32), Says(x11, x12, x13)) => false
  case (Says(x11, x12, x13), Gets(x21, x22)) => false
  case (Gets(x21, x22), Says(x11, x12, x13)) => false
  case (Notes(x31, x32), Notes(y31, y32)) =>
    Message.equal_agent(x31, y31) && Message.equal_msg(x32, y32)
  case (Gets(x21, x22), Gets(y21, y22)) =>
    Message.equal_agent(x21, y21) && Message.equal_msg(x22, y22)
  case (Says(x11, x12, x13), Says(y11, y12, y13)) =>
    Message.equal_agent(x11, y11) &&
      (Message.equal_agent(x12, y12) && Message.equal_msg(x13, y13))
}

implicit def equal_event: HOL.equal[event] = new HOL.equal[event] {
  val `HOL.equal` = (a: event, b: event) => equal_eventa(a, b)
```

```
}

} /* object Event */

object Lista {

def upt(i: Nat.nat, j: Nat.nat): List[Nat.nat] =
  (if (Nat.less_nat(i, j)) i :: upt(Nat.Suc(i), j) else Nil)

def drop[A](n: Nat.nat, x1: List[A]): List[A] = (n, x1) match {
  case (n, Nil) => Nil
  case (n, x :: xs) =>
    (n match {
       case Nat.zero_nat() => x :: xs
       case Nat.Suc(m) => drop[A](m, xs)
     })
}

def take[A](n: Nat.nat, x1: List[A]): List[A] = (n, x1) match {
  case (n, Nil) => Nil
  case (n, x :: xs) =>
    (n match {
       case Nat.zero_nat() => Nil
       case Nat.Suc(m) => x :: take[A](m, xs)
     })
}

def member[A : HOL.equal](x0: List[A], y: A): Boolean = (x0, y) match {
  case (Nil, y) => false
  case (x :: xs, y) => HOL.eq[A](x, y) || member[A](xs, y)
}

def map[A, B](f: A => B, x1: List[A]): List[B] = (f, x1) match {
  case (f, Nil) => Nil
  case (f, x21 :: x22) => f(x21) :: map[A, B](f, x22)
}

def equal_list[A : HOL.equal](x0: List[A], x1: List[A]): Boolean = (x0, x1)
  match {
  case (Nil, x21 :: x22) => false
  case (x21 :: x22, Nil) => false
  case (x21 :: x22, y21 :: y22) =>
    HOL.eq[A](x21, y21) && equal_list[A](x22, y22)
  case (Nil, Nil) => true
}

} /* object Lista */

object CocaineAuction {

import /*implicits*/ Nat.equal_nat, Event.equal_event

def CA_f0(t: List[Event.event], pS: Nat.nat, f: Nat.nat): Boolean =
  Lista.equal_list[Event.event](t, Lista.map[Nat.nat,
```

```
      Event.event]((i: Nat.nat) =>
                     Event.Says(Message.Server(), Message.Friend(i),
                                Message.Key(pS)),
                   Lista.upt(Nat.zero_nat(), f)))

def CA_fi(t: List[Event.event], pF: Nat.nat, f: Nat.nat, bi: Nat.nat,
           pS: Nat.nat):
      Boolean
  =
  Lista.equal_list[Event.event](t, Event.Says(Message.Friend(f),
       Message.Server(),
       Message.Crypt(pS, Message.MPair(Message.Key(pF), Message.Number(bi)))) ::
                                   Lista.map[Nat.nat,
        Event.event]((i: Nat.nat) =>
                       Event.Says(Message.Friend(f), Message.Friend(i),
                                  Message.Crypt(pS,
          Message.MPair(Message.Key(pF), Message.Number(bi)))),
                     Lista.upt(Nat.zero_nat(), f)))

def CA_fn(t: List[Event.event], pF: Nat.nat, f: Nat.nat, bi: Nat.nat,
           msg: Nat.nat):
      Boolean
  =
  Lista.equal_list[Event.event](t, Lista.map[Nat.nat,
      Event.event]((i: Nat.nat) =>
                     Event.Says(Message.Server(), Message.Friend(i),
                                Message.Crypt(pF,
        Message.MPair(Message.Number(bi), Message.Number(msg)))),
                   Lista.upt(Nat.zero_nat(), f)))

def winner_i(x0: List[Event.event]): Option[Nat.nat] = x0 match {
  case Nil => None
  case x :: evs =>
    (x match {
       case Event.Says(_, Message.Server(), Message.Agent(_)) => None
       case Event.Says(_, Message.Server(), Message.Number(_)) => None
       case Event.Says(_, Message.Server(), Message.Nonce(_)) => None
       case Event.Says(_, Message.Server(), Message.Key(_)) => None
       case Event.Says(_, Message.Server(), Message.Hash(_)) => None
       case Event.Says(_, Message.Server(), Message.MPair(_, _)) => None
       case Event.Says(_, Message.Server(), Message.Crypt(_, a)) =>
         (a match {
            case Message.Agent(_) => None
            case Message.Number(_) => None
            case Message.Nonce(_) => None
            case Message.Key(_) => None
            case Message.Hash(_) => None
            case Message.MPair(Message.Agent(_), _) => None
            case Message.MPair(Message.Number(_), _) => None
            case Message.MPair(Message.Nonce(_), _) => None
            case Message.MPair(Message.Key(aa), _) => Some[Nat.nat](aa)
            case Message.MPair(Message.Hash(_), _) => None
            case Message.MPair(Message.MPair(_, _), _) => None
            case Message.MPair(Message.Crypt(_, _), _) => None
```

```
            case Message.Crypt(_, _) => None
          })
       case Event.Says(_, Message.Friend(_), _) => None
       case Event.Says(_, Message.Spy(), _) => None
       case Event.Gets(_, _) => None
       case Event.Notes(_, _) => None
     })
}

def CA_i(l: List[Event.event], pF: Nat.nat, f: Nat.nat, b: Nat.nat => Nat.nat,
          pS: Nat.nat, x5: Nat.nat, kl: List[Nat.nat]):
      Boolean
  =
  (l, pF, f, b, pS, x5, kl) match {
  case (l, pF, f, b, pS, Nat.zero_nat(), kl) => CA_f0(l, pS, f)
  case (l, pF, f, b, pS, Nat.Suc(i), kl) =>
    CA_fi(Lista.take[Event.event](Nat.Suc(f), l), pF, f, b(Nat.Suc(i)), pS) &&
      (winner_i(Lista.drop[Event.event](Nat.Suc(f), l)) match {
         case None => false
         case Some(k) =>
           Lista.member[Nat.nat](kl, k) &&
             CA_i(Lista.drop[Event.event](Nat.Suc(f), l), k, f, b, pS, i, kl)
       })
}

def CA_n(t: List[Event.event], pF: Nat.nat, f: Nat.nat, pS: Nat.nat,
          b: Nat.nat => Nat.nat, n: Nat.nat, msg: Nat.nat, kl: List[Nat.nat]):
      Boolean
  =
  CA_fn(Lista.take[Event.event](f, t), pF, f, b(n), msg) &&
    (Nat.less_nat(Nat.zero_nat(), n) &&
      CA_i(Lista.drop[Event.event](f, t), pF, f, b, pS, n, kl))

} /* object CocaineAuction */
```