Saarland University
Faculty of Natural Sciences and Technology I
Computer Science Department

Bachelor

Caspa - Mechanized Verification of Security Protocols Using Causality-based Abstractions

submitted by

Stefan Lorenz

May 7, 2009

Supervisor	Prof. Dr. Michael Backes
Advisors	Prof. Dr. Michael Backes
	Dr. Matteo Maffei
Reviewers	Prof. Dr. Michael Backes
	Dr. Matteo Maffei

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, May 7, 2009

Stefan Lorenz

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, May 7, 2009

Stefan Lorenz

Acknowledgment

I would like to thank Prof. Dr. Michael Backes for supervising and advising my work and issuing this interesting topic to me. I also thank Dr. Matteo Maffei for advising my work and proofreading my thesis. Both have given me deep insights on countless occasions. Also many thanks go to Kim Pecina for proofreading my thesis. Finally I would like to thank all my fellow students whose company and advice I learned to appreciate and my family for the continuous support.

Abstract

There exist a lot of cryptographic protocols that claim to fulfill goals such as keeping a message secret or providing authenticity. Time has shown that even if we assume the cryptographic principles secure, a protocol can be flawed by design. It is important to use formal methods to prove the claimed security guarantees of a protocol because intuitive reasoning often misses important cases. This work addresses the automated verification of safety properties of cryptographic protocols.

Caspa, the tool we present, is capable of verifying authenticity of participants as well as the secrecy of messages in a fully automated way for a possibly unbounded number of parallel sessions. It is based on the notion of causal relations between messages and events in a protocol introduced in [3]. Caspa offers an easy-to-use graphical interface that allows even users not familiar with the underlying theory to analyse the security of a protocol.

This work will present the analysis technique as well as the architecture of the tool, focusing in particular on the implementational details that make features as attack reconstruction possible and the different ways to interact with the program.

Table of Contents

Section 1

Introduction

1.1 Motivation

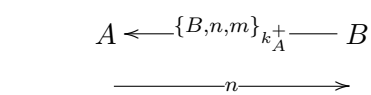
Communication is a daily activity in our lives. We talk to people, send emails, write letters, do phone calls, etc. All these forms of communication follow certain rules. When talking to someone we meet on the street, we usually start the conversation by greeting him. Afterwards the real conversation by means of exchanging information begins and at the end we say goodbye to each other. If we write these simple rules down and follow them for every conversation of this type we do, then we stated a protocol. A protocol defines the "rules of conversation" for a specific communication. On the Internet we use protocols for nearly every action we do without really noticing it: Http when visiting a webpage, ftp when downloading a file, and smtp for sending emails to name a few.

In the real world we usually can see the person we talk to and thus decide whether it is the person we think it is and we can check that no one else is eavesdropping on our conversation. When we talk to other people over the Internet a problem arises. Usually we have now idea whether the person we are talking to is the one whom he claims to be and we do not know whether the messages we send are unchanged or kept secret. This issue is addressed by deploying cryptography into protocols to ensure goals such as keeping a message secret or authenticating someone. In this case we talk about cryptographic protocols.

In order to standardize protocols and make them available for everyone, they are written down in some representation. We will introduce a graphical one that is easily comprehensible for humans and a more technical representation, the common syntax [10], [8], that allows for machine processing, and use them for the rest of this work.

```
1:  B -> A:  {B, n, m}PK(A)
2:  A -> B:  n
```

protocol in common syntax



protocol in graphical representation

The goal of the protocol above is to establish a secure and authentic exchange of the message `m`. In other words, after running the protocol, the message `m` is only known to the participants `A` and `B` and at the end `B` is convinced that the message was received by `A`. The first part of a step in the protocol in common syntax is of the form `A -> B:`, meaning that `A` is sending a message to `B`. In literature `A` and `B` are most often referred to as `Alice` and `Bob`

and we will follow this through the remainder of this work. Other parties are `C` as the general attacker and "man-in-the-middle" and `S` a (trusted) server. Then, after a colon, follows the information that is sent in this step. In the graphical representation the parties are the nodes of a directed multigraph. The edges are communications, content is denoted on the edge. It points from the originator to the receiver.

The first message sent in the example is a ciphertext, indicated by the curly brackets around it, that is encrypted with the public key of `A`, denoted by `PK(A)` or $k_A{}^+$. This first message is called the challenge. The ciphertext contains the identity `B`, a nonce `n`, and the message `m`. In the second step of the protocol, the response, `A` returns the nonce received in step one to `B` and the protocol is finished. We assume that public keys are known to everyone without explicit distribution. Moreover we assume that cryptography used in the protocols is perfect, or at least as good as we want it to be. These are abstractions from the real world, but we want to reason about the logic of the protocol, not the security of concrete algorithms or even the question whether concrete algorithms that provide certain guarantees exist at all. The goal is to decide whether the protocol could fulfill the goals it claims to address under the assumption that the cryptography involved works. From this assumption follows directly that only `Alice` can decrypt the ciphertext received in the first part with her private key that only she knows. Thus the security of the message `m` is given. It is sent only once in encrypted form.

The second goal of the protocol is that `B` can be sure that `A` actually received `m`. At this point, the nonce `n` gets important. A nonce is a fresh value, usually a number, that is only used once. It is generated randomly every time it is used, so no attacker can guess the value of a nonce. Due to the fact that the nonce `n` is encrypted together with the message `m`, only `Alice` can decrypt it and so she is the only one besides `Bob` who knows this value. When `Bob` receives the nonce `n` he can be sure, that `Alice` was the sender of the response. We say `B` authenticates `A`.

Can we do any better? Protocols are designed to fulfill tasks in the real world. There the time needed for encryption and decryption and the amount of bytes a message sent over the Internet consists of are important factors. Additionally we assume that every message sent in a protocol step contains some information about its origin, so `Alice` still knows where the message she receives comes from, even without the explicit use of the identity in step one of the protocol. So we could try to leave out the identity `B` in the first step of the protocol.

Our modified example looks as follows:

```
1:  B -> A:  {n, m}PK(A)
2:  A -> B:  n
```

changed protocol

But now a problem arises. Like in the real world, where we have for example ip-addresses to determine the origin of a message, we can't trust this embedded origin information. Malicious parties are able to spoof them as they are with ip-addresses. So `A` cannot be sure any longer that the message she receives originates from the party that claims to be the originator. These dishonest parties could try to inject information like viruses or spyware into the communication by convincing `A` to execute the message she believed came from `B` or just try to get information out of the conversation of third parties. In our abstraction we denote these malicious parties as `E` and call it the environment. Now let us have a closer look at what

happens if we leave out the explicit origin information.

In the first step we still send m and the nonce n encrypted with the public key of A but we leave the identity B out. This opens the door for an attack. E now can intercept this first message, change its origin to be him and then forward it to A. Alice is now convinced that the protocol has been initiated by E whereas B thinks he is talking to A. A then decrypts the ciphertext and returns n to E. E forwards it to B and the protocol run is finished. The goal of authenticity is broken. Alice is convinced, she has a protocol run with E and B is convinced to have a protocol run with A. However the message m remains secret. E has interfered with the protocol run but still knows nothing about m. So what is the problem about E breaking authenticity? This will get clear when we take a look at the following example for a possible usage of this protocol.

Consider A as some institution that sends confidential information out to people if it receives a request via the Internet using our protocol. The message m specifies what kind of information is requested. The attacker does not learn m but still A would send any information to him because he was the originator of the request. So even without revealing the message m this breach of authenticity renders our protocol insecure.

Here we reach the borders of intuitive reasoning. Even this small change, that looks harmless on first sight, breaks one of the goals of the protocol. If we consider bigger protocols with even four, five or more steps, it gets much more complicated. An attacker can try to use messages from one protocol session in each other. We will demonstrate this on a second example, the Needham Schroeder Symmetric Key protocol [14]. The goal of this protocol is to exchange a symmetric key between Alice and Bob using a trusted server S with mutual authentication between A and B. In this protocol a special operation dec is used. This operation decrements a nonce. This is possible as we have seen that nonces are usually numbers.

1: A -> S: A, B, na
2: S -> A: $\{na, B, K_{AB}, \{K_{AB}, A\}K_{BS}\}K_{AS}$
3: A -> B: $\{K_{AB}, A\}K_{BS}$
4: B -> A: $\{nb\}K_{AB}$
5: A -> B: $\{dec(nb)\}K_{AB}$

Needham Schroeder symmetric key protocol

Both parties, A and B, have a preshared key with the trusted server S, namely K_{AS} and K_{BS}. In the first step of the protocol Alice sends her identity, the identity of the person she wants to communicate with and a nonce to the server. The server then creates a session key for the parties that are willing to communicate and creates a message designated for the intended communication partner B. He returns this message, encrypted with his shared key with B, the session key, the nonce of Alice and the identity of the intended communication partner, B, all encrypted with his shared key with A. The encrypted pair of the session key and A's identity can only be decrypted by the designated communication partner B. Thus a malicious originator cannot change or modify the content of this package in order to get B talking to someone else. Alice now forwards this package to Bob. He then knows, after decrypting it, that it is Alice who started the conversation by her identity and receives the session key for their further communication. Again the fact that S is a trusted server is important. The ciphertext Bob receives in step 3 is encrypted with the key he shares with the server. So Bob is convinced that the server really got a request that Alice wants to communicate with

him. B then sends A a nonce encrypted with the freshly distributed session key. Thus Alice knows that Bob received her request and got the session key out and she is convinced that she is really talking to Bob. In order to convince Bob that she is the one whom she claims to be, by proving to know the session key, she decrements the nonce received from Bob and returns it to him, encrypted with the session key. This change in the content is important because without this an attacker could just mirror the message sent in step 4 to Bob breaking the mutual authentication. After receiving the decremented nonce in the last step of the protocol, B is also convinced that he is talking to Alice. Both now share a session key and authenticated each other.

However this protocol suffers from a weakness. We have to assume that an attacker can record every step of the protocol. He may not be able to decrypt anything, but he can try to insert it in a parallel or later protocol session. Furthermore a session key, as K_{AB} in this example, could become compromised after the session and thus should never be used twice, as the name indicates. A session key could get lost by means of social engineering or weaknesses on the machine it is used, or maybe it is just a weak key that is only strong enough for a certain amount of time. As soon as this happens the authentication of the protocol can be compromised. Step 3 of the protocol contains the session key and the initiator of the conversation, but no freshness information like a timestamp or a nonce. A malicious party can resend step 3 of the protocol to Bob in a later or parallel session, after compromising the session key.

1: A -> S: A, B, na
2: S -> A: $\{na, B, K_{AB}, \{K_{AB}, A\}K_{BS}\}K_{AS}$
3: A -> B: $\{K_{AB}, A\}K_{BS}$
3a: E -> B: $\{K_{AB}, A\}K_{BS}$
4a: B -> E: $\{nb\}K_{AB}$
5a: E -> B: $\{dec(nb)\}K_{AB}$
Needham Schroeder symmetric key protocol attack

In step 4a Bob would send the message to Alice but E intercepts it. Intercepting messages and simply dropping them is something we assume an attacker can always do. At the end of this modified protocol run, Bob is convinced he is talking to Alice but in fact he is talking to the attacker. Since the attacker compromised the session key, he is now able to decrement the nonce sent by Bob and thus send him the correct response in 5a. After the protocol run, B could use the now shared session key K_{AB} for further communication, so in the latter case B would communicate with E as he would be a trusted party.

The protocol was proposed in 1978 and it took three years until the attack was found by Denning and Sacco [9] and this is still a protocol of moderate complexity. In the first example we saw that leaving a single identity out can give an attacker the option to change the meaning of a message without even breaking its secrecy. The second one made it clear that even a protocol that looks secure can break down if we take parallel sessions into account. Moreover we noticed that it took years to discover the weakness in the latter example. So we have to have formal methods for analyzing protocols in order to be able to handle these huge amount of completely different possible weaknesses of protocols.

1.2 Analysis techniques

First of all the bad news: analyzing safety properties of cryptographic protocols is undecidable. Intuitively this is because of the fact that an attacker could run arbitrary many protocol sessions in parallel at the same time to attack one single session. In order to decide whether the protocol is safe or not, we would have to use a machine that takes into account an unbounded number of protocol sessions in finite time. There exists no such machine. However we can still reason about cryptographic protocols and make statements about their security with certain restrictions. In general there are two approaches to this problem, dynamic analyses and static analyses.

Dynamic analyses try to explore every possible state of a protocol by running it. These techniques are also called model-checking or state-space exploration. If it is possible to cover every possible state when model-checking, this technique can actually proof or disproof security of the checked model. When talking about hardware, that has a finite set of states, this works fine, given that we have enough resources to explore every state. For analyzing protocols this does not hold. As mentioned, an attacker could use arbitrary many protocol sessions in parallel to attack one. This cannot be modeled with a finite state set. We can still try to cover as many states as our resources allow. If a flaw is found we can often exactly determine where it occurred and how. However, if no flaw was found, this can either mean we did not spend enough resources on searching, or there is none. So dynamic analyses can proof weaknesses and give hints on the security of a protocol.

Static analyses on the other hand do not actually run the protocol. Here we reason about an abstraction of the actual protocol semantics, i.e., we abstract away from exactly these aspects of protocols that make them undecidable. As result we do not have to explore an infinite state space any more.

The drawback is that we lose precision due to the abstraction. Everything that can be proven in the abstraction holds in the real model too but the opposite does not hold. If we cannot deduce the security of a protocol in our model, this can either be a real flaw or it may be that the abstraction is too unprecise to analyze it correctly. However with static analyses we can actually prove a protocol secure by proving it secure in the abstraction.

Section 2

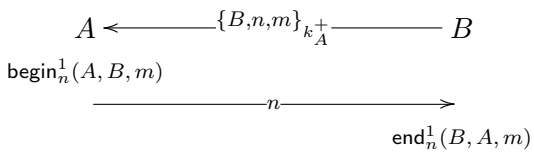
Causality-based abstraction

Caspa implements a static analysis called causality-based abstraction introduced in [3]. The idea behind this technique is to capture the causality between events and messages sent in a protocol. Basically this is done in two steps: First the protocol is translated into a causal graph $\mathscr{C}$. This is a finite directed graph that models the causal relations occurring in a protocol. The nodes or abstract processes represent states the protocol can be in. These processes are so-called ρ-spi processes. If we follow the edges from a given node v up to a source of the graph we can see which states must have occurred in order for a session to reach this state. We talk about a trace set or path. There can be more than one path for a given node v. The second step of the technique is to check all these trace sets capturing the causal relations needed to reach a specific state separately for their safety. It is sufficient for a path to be safe if there is at least one safe trace in the set, but for a node v to be safe all pathes that lead there must be safe.

One of the major benefits of causality-based abstraction is that arbitrary many protocol sessions are abstracted into one finite graph. This gives us the possibility to decide whether our model is safe or not. So if we can prove a protocol safe with this technique, it is safe, even if run with unboundedly many sessions in parallel.

2.1 ρ-spi calculus

In order to be able to translate our protocol into a causal graph we have to represent it in a dialect of the ρ-spi calculus from [6, 7]. It is derived from the spi-calculus introduced in [1]. While the spi-calculus is appropriate for the analysis of authentication protocols, the extension introduced in ρ-spi associates principal identities with processes, binds keys to their owners and contains authentication-specific constructs. As mentioned above, every node represents a process. The source of the graph is a process representing the whole protocol. Such a process is a chain of instructions that are processed one by one, e.g. begin authentication - read a nonce - end authentication. Two processes running at the same time are called threads. In order to go from one process to another we have to reduce it. Reducing here means abstractly executing the first instruction in the chain and moving on to the resulting process. One possible reduction is called intra-thread reduction and can be done without interacting with another thread. The other one is called inter-thread reduction. This, means that in order to reduce a process, input from another process is needed. In this way communication is modeled. Intra- as well as inter-thread reduction can cause a split into several threads.




```
init := new(m).new(n).out({B, n, m | pubkey(A)}).in(n).end(1 n B A m).0;
resp := in({B, x, z | seckey(A)}).begin(1 x A B z).out(x).0;

newkey(A).(B |> init || A |> resp)
```

Figure 2.1: Protocol with authentication assertions

The former if we reach a `parallel`-instruction, the latter if there is more than one possible communication for an `in`-instruction.

Now we will present the example from the first chapter, translated into the ρ-spi calculus. Afterwards we will see the grammar of the dialect introduced in [3] in a slight variation suitable for automated processing by Caspa.

The first line describes the actions done by the initiator of the protocol, `Bob`. Assigning this part of the protocol to the label `init` and later on using this label instead of the whole first line allows us for a clean way of writing of protocols. So `Bob` first creates the names `m` and `n` with the `new`-instruction. The abstraction does not differentiate between nonces and messages, both are names and are unguessable by an attacker. All names used have to be introduced by a `new`-instruction because the only entities known to the participants at the beginning are public keys, their secret keys, symmetric keys, and identities. The latter have not to be introduced by `new`. A name that is not introduced by a `new`-instruction is free and hence known to the attacker. If the same `new` happens to be visited more than once, for instance by an attack in a parallel session, the new name gets α-renamed and is treated as being different from the one already introduced. This enlarges the graph dramatically as we will see later on, but makes our analysis much more precise. There are protocols we could not analyze without α-renaming.

Free names are abstracted to `EPS` when the protocol is read. `EPS` abstracts every message the environment can invent by itself including every identity possible and public knowledge.

`Bob` then sends his identity, `m` and `n` encrypted with the public key of `Alice`. The whole ciphertext enters the knowledge of the environment. The environment contains everything that is public knowledge and thus everything that an attacker could deduce.

The second thread, labeled with `resp` for response, begins with the reading of a ciphertext encrypted with the public key of `A`. There are variables in this `in`-instruction that will bind to the input read. Caspa only considers messages that fit the pattern of the `in`, e.g. if a triple encrypted with the secret key of `Alice` is read then only an encryption of a triple with her public key will be considered a match. As mentioned above communication has no specific target so the environment simply tries to plug everything into this `in`-instruction that fits the pattern. This includes of course the correct message sent by `B`, but also every triple that can be constructed out of the knowledge the environment has learned so far, including `EPS`.

It is necessary for the environment to exploit all possible matches for an `in` in order to show that no maliciously crafted message inserted instead of the honest one leads to an unsafe path and thus can break authenticity or secrecy of the protocol. So only if the environment

Name

a	::=	I, J, A, B, E	identities
	|	n, m	messages
	|	symkey (I, J)	symmetric keys
	|	pubkey (I)	public-keys
	|	seckey (J)	secret-keys
	|	epsilon	environment messages

Process

P, Q	::=	new(n).P
	|	newkey(I).P
	|	in(M).P
	|	out(M).P
	|	begin(i n A I M).P
	|	end(i n A I M).P
	|	A |> P
	|	P || Q
	|	0[2]

Term

M, N	::=	a	names
	|	x, y, z	variables
	|	M,N[1]	tuples
	|	{M | u}	ciphertexts

Notation: u ranges over names and variables

Table 2.1: Grammar of the ρ-spiCalculus as accepted by Caspa

creates all possible matchings for this `in`, including the honest communication, we can be sure that we can analyze the protocol correctly.

In the example a possible match for the `in` is the ciphertext sent by `B`. So the nonce `n` will be bound to the variable `x` and the message `m` to the variable `z`. This is denoted by labeling the names with the variables yielding $\mathtt{n}_{\mathtt{x}}$ and $\mathtt{m}_{\mathtt{z}}$. If a labeled message is then send out again and bound to a new variable, the old labeling is overwritten.

The next instruction, the `begin`-assertion, is one of the extensions of the ρ-spi calculus. This denotes the beginning of the authentication of `Alice` on the reception of `m` using the nonce `n` with `Bob`. The numbers indicate what `begin` corresponds to which `end`. Afterwards `A` sends out the nonce `n` and then the thread stops with `0`, the `stop`-instruction. In the other thread `Bob` expects the `n` as response and on receiving `n` proceeds with the `end`-instruction corresponding to the `begin` in the `resp`-thread authenticating `Alice` receiving `m` in a session using nonce `n`. The identities in this `end` are swapped compared to the `begin`. The first one is the party acting, i.e., starting or finishing an authentication and the second one is the party interacted with. Then this thread reaches its end, too.

The third line describes how the threads, now assigned to labels for easier use, work together. `A |> resp` denotes that `Alice` starts the process `resp` and `B |> init` that `Bob` executes `init`. This happens in parallel indicated by `|` between the two statements. `|>` is another extension of ρ-spi associating principals with processes. Before the two threads start, the public key of `Alice` enters public knowledge by the `newkey`-instruction.

The full grammar of the dialect of the ρ-spiCalculus used in Caspa is summarized in Table 2.1.

Everything starting with a capital letter will be seen as an identity by Caspa, while `E`

[1] For the sake of readability Caspa will explicitly not accept parentheses around tuples.
[2] Although we write 0, we say `Stop`.

denotes the adversary. Strings starting with the small letters "x", "y" and "z" indicate variables and everything else starting with a letter is a message. The message `epsilon` abstracts every message generated by the environment, i.e., identities, public keys, and attacker keys. Caspa uses `EPS` in the graph and $\mathscr{E}$ in the traces to denote `epsilon`.

2.2 Causal graph generation

The top node of the graph is always a node containing the process representing the whole protocol. The first part of the process is then examined and children added accordingly. As mentioned above there are two types of causalities, namely intra-thread and inter-thread causality. For every process `q.P` where `q` is not an `in`-instruction we have intra-thread causality. Let v be the node representing `q.P` then we create a new node as child of v with process `P`. There are some processes that are treated specially, i.e., parallel execution, principal and stop. For parallel execution two children, one for every process, are added. Principal results in a child `P` and stop is the last node without any children. Reducing `in-` nodes requires a message that matches the pattern. There can be more matches than the one intended by the protocol run. The environment will plug in everything that fits. For every possible message a child is created with a special link in between, a synchronization point, indicating the communication and connecting the sources of the message with the `in`-node and the resulting child. We will revisit them when we explain the edges in the next paragraph. Now consider the process `in(x, y).P` as an example. Only the message `m` entered the knowledge of the environment so far. So there would be four possible matches for this `in`-node, namely (`m`, `m`), (`m`, $\mathscr{E}$), ($\mathscr{E}$, `m`), and ($\mathscr{E}$, $\mathscr{E}$), resulting in 4 new nodes. This is the reason why α-renaming can have a heavy impact on the size of the graph as mentioned above. Suppose the message `m` would be recreated resulting in `m'`. Then the possible matches would grow from four to nine and thus further enlarge the graph. This growth is exponential.

While nodes represent states of the protocol, the edges represent the causality between them. Intra-thread causality is represented by a single edge pointing from the predecessor node to the successor. For inter-thread causality, things are slightly more complicated. As mentioned above there exists a synchronization point linking the `in`-node, the sources of the message matching the pattern and the successor of the `in`-node together. The point as well as the links to the sources are labeled. The former with the actual message matching the pattern and each of the latter with the integer-component. Intuitively the integer-component determines which component of a message necessarily belongs to one single session. This is important as the following example shows. Consider the name `n` sent out over the network. If this name now matches two different `in`-nodes we cannot be sure that the name in both nodes was taken from the same session as we abstract away from different sessions run in parallel. This is addressed by the integer-component. Another example for an integer-component is a ciphertext where the environment does not know the key. Without the key the content of this message has to originate from the same protocol session, the one the ciphertext originates from.

2.3 Analysis - secrecy and authenticity

After constructing a causal graph we can begin the analysis. The two properties we want to prove are secrecy and authenticity. The first is relatively easy to check. A name `n` is secret

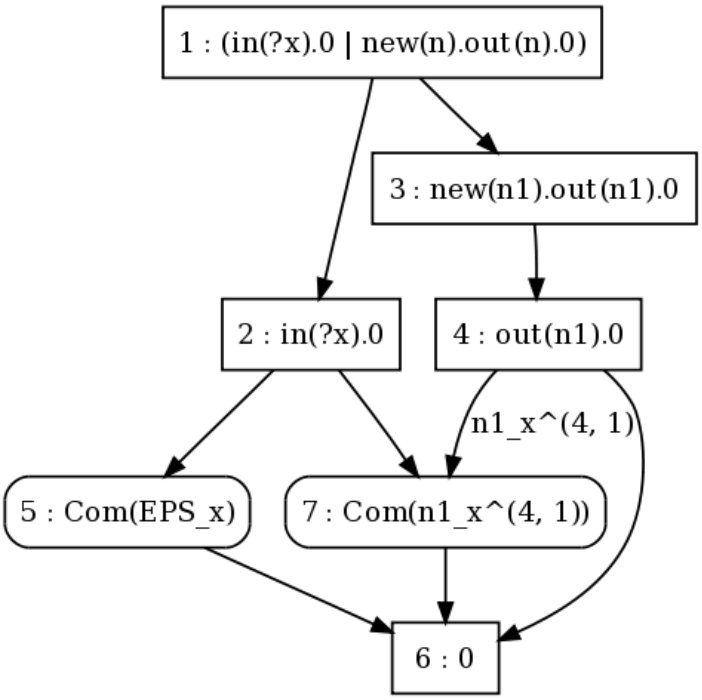


Figure 2.2: Example of a causal graph

if it never enters the knowledge of the environment at any point. Usually this goal is only intended for some names occurring in a protocol, e.g. secret keys or specific messages. In our example the message `m` is secret but the nonce `n` is not. `m` is only sent encrypted and the secret key of `A` never leaks. `n` however is returned in plain by `A` and thus enters the environment.

To check authenticity we have a little bit more work to do. First of all authenticity cannot be checked if there are no `begin`- and `end`-assertions annotated in the protocol. If so, there has to be one matching `begin` in every path leading to one single `end`. Intuitively, whenever a party finishes authentication with another party, the latter initiated it beforehand. The environment however must not use assertions because then every protocol can be rendered insecure. Consider for example the null process `0`, that, by itself, is perfectly secure. However, the protocol `0 || E |> end(1 EPS E A EPS).0` would not be safe. Moreover we do not want to check whether the environment can authenticate with an honest principle.

Now we have to take a closer look on paths as we need them for checking authenticity. A path is a set of causally related traces needed to reach a designated node v in the graph $\mathscr{C}$. A trace is what one usually calls path when talking about graphs. It is one single way from the source to v. To clarify this concept we introduce a small example. Figure 2.2 is the causal graph for the protocol `in(x).0 || new(n).out(n).0`. The superscripts at the synchronization points denote the origin node of the integer component by the first number and the positional index it was sent at by the second number. All nodes in our graph have a unique id, so a number is sufficient for determining the origin. We see these ids in Fig. 2.2 in front of the protocols of the nodes. This indexing is important when proving equality of names later on. The numbers behind newly created messages originate from α-renaming.

As an example for trace and path generation we will compute the paths for the `Stop`-node,

$$\left\{ \begin{array}{l} new\,(n1) :: out\,(n1) :: in_{com}\left(n1^{4,1}_{(x)}, n1_{(x)}\right) \\ in\left(n1_{(x)}\right) \end{array} \right\}$$

$$\left\{\ new\,(n1) :: out\,(n1)\ \right\}$$

$$\left\{\ in\left(\mathscr{E}_{(x)}\right)\ \right\}$$

Figure 2.3: Paths of `Stop` for causal graph 2.2

the one with the id 6. When creating the paths for the node, we start with the node and follow the graph up towards the source. In the following we will refer to the nodes by their id for better readability. Node 6 has three parents, namely 5, 4 and 7. These nodes are causally unrelated so we will have at least three paths. The only parent of 5 is 2 whose only parent is 1, the source of the graph. This gives us our first trace 1-2-5-6. There is no causal relation in any of the nodes of this trace so this is already the first path consisting only of a single trace. This trace is sufficient for reaching the `Stop`-node. Following the node 4 directly yields the trace 1-3-4-6 as is again a direct path as there is no branching here either. The last parent of node 6 is 7. This node represents a synchronization point. It has two parents, an `in` and a corresponding `out`. So there is a causal relation and we have to follow both parents resulting in two traces needed to reach the top via node 7. The one is 1-3-4-7-6 and the other 1-2-7-6. Both of them form the third path and as there is no more branching on the way up we got all paths for the node 6. In general, whenever there is a synchronization point, the number of traces in one path increases because more than one trace is needed to reach the child, whenever we have a branch that is not the result of a communication, the number of paths increases. An exception is the communication where everything sent is created by the environment. In this case no `out`-node is required to reduce the `in`. For every other inter-thread causality there is one trace denoting the reduction of the `in`-process and one trace for every `out`-process used. The resulting paths for node 6 are depicted in Fig. 2.3.

Paths are not only a formal construct for analyzing the authenticity of our protocols, they can also give hints towards the cause of possible flaws. Caspa will output the path leading to a break of authenticity as well as all pathes created. Following the traces in the path, it can be checked whether it is an actual flaw and if so, at what point of the protocol and how it occurred.

The description of trace finding is close to the implementation in Caspa. Real traces are not represented by numbers but by actions. Actions are abstractions for the events that take place in the real protocol. An `out`-node for example is transformed into an `out`-action that represents a principal sending a message over the network. Neither parallel execution nor `Stop` are represented by actions. Moreover, when computing the traces for a node v the traces are computed up to this node, so the action for v is not part of the trace. A special case are `in`-nodes followed by a synchronization with `out`-nodes. Then the `in`-node is transformed into an `in`-action representing the event that something is received and an `incom`-action that follows the `out`-action of the corresponding `out`-node. The `incom` contains the integer component as first element and the message matching the pattern as second component. The indexing is the same as we have seen in the causal graph above.

However there is one more thing that has to be checked before we can prove authenticity. The names appearing in a causal graph do not necessarily originate from one single session, but when arguing about authenticity the names in the corresponding assertions have to. In order to proof the graph cycle-invariant we have to proof that we do not lose equality of names, i.e., that names, which are equal when running through a cycle once, still are equal when running through it several times. This is important to consider as the abstraction only visits each cycle once. Only if the graph is cycle-invariant we perform a check for authenticity.

Proving two names in a causal net equal is done using a deduction system. Intra-thread reduction has no impact on the equality. As there are no new names introduced all equalities before the reduction still hold afterwards. When it comes to inter-thread reduction it gets more difficult. We mentioned above that integer components guarantee origination from one single session. So names from integer components are the only ones that can be proven equal for inter-thread causality, as names, that are not from integer components and lost equality, can still become equal to a name appearing earlier in a trace, if they are pattern matched.

Once proven cycle-invariant we can check the graph for authenticity. In order to do so we compute the paths for every `end`-assertion in it. We consider a graph providing weak authenticity if it is cycle-invariant and in every path for every `end`-assertion exists at least one trace containing a matching `begin`-assertion. It provides strong authenticity if there is exact one matching `begin`-assertion in every path.

Section 3

Architecture

We decided to write Caspa in Ocaml [13], a functional programming language with object oriented and imperative extensions. This gives us the power of implementing most of the definitions "as is" as functions but we can also use object oriented principles for implementing own graphs or trees for example. Moreover we have the typical features of functional languages like compile time type checking and a minimum of runtime errors. Especially when it comes to mathematical definitions functional languages tend to produce very readable and clean code in comparison to imperative languages.

We split the source code into logical groups of functions gathered in compilation units. Every such module got its own file together with an interface definition. We tried to follow the Ocaml implementation in this as much as possible for optimal re-usability and compatibility of the functions we wrote.

The implementation of Caspa follows closely the technique presented in [3]. First the protocol will be translated into a graph, then the paths and path sets are computed. Afterwards secrecy and authenticity are checked. We will follow this line in the description of the architecture as well. Due to the fact that Caspa reached a complexity that makes it impossible to be presented completely here, we will focus on the key algorithms, i.e., graph generation and path generation. Additionally we will explain the three extensions Thread Coloring, Secrecy Tracking and Attack Reconstruction, which improve the usability of the output of Caspa for further analysis greatly with only a minimal computational overhead.

Another interesting part of Caspa is the way users can interact with it. We will discuss this in detail in the next chapter.

3.1 Graph generation

The first problem was the representation of the graph in the program. We decided for an object oriented approach using a class representing the nodes. This gives us complete flexibility for inserting, deleting and exchanging nodes in the graph. The node itself stores every important information, e.g. the protocol it represents, and in addition we gave the node some useful functions like one for printing itself. A special attribute of nodes worth mentioning separately is their id. Whenever a new node is created, it automatically receives a unique id. Using this id we can easily identify and compare nodes in the graph. When we output the graph the nodes are annotated with their id. As in the Ocaml implementation we hide the concrete implementation by a module with functions that operate on nodes. Only these functions are

used throughout Caspa. This allows for exchanging the implementation of the nodes without changing any project using them. Edges are modeled implicitly by the predecessor links in the nodes. Every node has got a reference to its parent nodes. This is sufficient as the trace generation works "bottom-up" as we will see in section 3.2. The complete graph is stored as a list of the nodes it contains.

The second problem was the question how to create the graph. Our first approach was a depth first search. We examined one branch of the protocol until we reached the `Stop`-node. Whenever we encountered a communication, everything that could be used for further reduction, was examined. This was easily implementable as a recursive function, moderate in memory consumption but for bigger protocols with lots of communication this algorithm needed an ever increasing amount of time because of the large computational overhead. The approach we finally chose can be compared to a breadth first search. We compute one reduction step at a time for all branches that may exist. This needs some more space to store the state we are in and some imperative programming constructs like loops, but on the other hand it terminates and Ocaml gives us the flexibility to use such structures by default.

Implementation

After the protocol is parsed, Caspa has to build a graph from it. This is implemented in the algorithms build_new_graph (Fig. 3.1) and step (Fig. 3.3). First we create the level list and initialize it with a node containing the whole protocol. Additionally an empty list for the `in`-nodes is created and the environment reset. An `in`-node in this context is a node containing a protocol beginning with an `in`-instruction. In the rest of this work we will refer to such nodes as `in`-nodes. The main part of this function is a loop running as long as there are nodes to reduce in the level or there are new `in`-nodes added. Inside the loop the step function for the current level is called. As long as there are new `in`-nodes added there might occur new branches in the graph so we have to investigate them and of course as long as there are still nodes to reduce too. This implements the closure operator on graphs from [3]. Intuitively we continue as long as there are changes.

For practical reasons it can occur that the level list still contains a node, but there are no changes any longer. This is not considered an error if the last node remaining in the list is the `Stop`-node. Therefore the condition of the while loop performs an additional check for the head of the level list not being the `Stop`-node.

α-renaming

An interesting detail you can see in the code listing 3.1 is the use of the `get_alpharenamed` method on the protocol to create the top node. Basically the author of a protocol can use the same name, e.g. `m`, in every thread he creates, but this would make it impossible for Caspa to distinguish where a message actually came from. To circumvent this problem we α-rename the protocol, i.e., we ensure that every `new`-instruction creates a syntactically unique name. This gives the author the freedom to re-use names in every thread. The actual implementation of `get_alpharenamed` inspects the first process in a given protocol and if it is a `new`-instruction the message is padded with a unique number and the rest of the protocol is capture free α-renamed accordingly. The method is applied whenever we add a new node to the graph.

This becomes especially interesting when we encounter a path in the graph containing a `new`-instruction several times. If the pathes behind the `new`-node are identical up to α-

```
let build_new_graph protocol = (
    let in_nodes = ref [] in
    let top = new node (check_restricted protocol;
        get_alpharenamed prot) true in
    let level = ref [top] in
    let old_in_size = ref 0 in

    (reset_environment ();
     nodelist := [top];
     level := step in_nodes !level;

     while List.length !level > 1 || List.length !in_nodes > !old_in_size
         || try protocol_compare ((List.hd !level)#get_protocol) Stop != 0 with
            _ -> false
     do
        level := step in_nodes !level
     done;
    )
)
```

Figure 3.1: Main loop for graph generation

renaming, no renaming will take place, but if they are different there will be a new `new`-node created and then α-renamed. This, however, enlarges the graph, but is essential to analyze protocols correctly, because in this case the messages are from different sessions and thus should be different in the graph. Fig. 3.2 shows an example of a simple protocol that creates a message after receiving one and then sends them both out. An attacker can feed the message created back into the protocol which yields a different run with a different message created. This new message can again be used as input. In the first run we send out the `Epsilon` message received together with the first created message. In the second run it is the newly created message together with the second created message. Afterwards every run would be semantically the same, i.e., last message together with new message. Therefore no more `new`-nodes are created and thus no new branches introduced.

Step Function

The step function computes the actual reduction for every node in the current level list. It iterates over all nodes in the list provided as argument and checks for every node whether it is an `in`-node or not. In the former case, it is added to the `in`-node list and all possible reductions for this `in`-node at this point of the program are added to the result. The next level is exactly the list returned by the step function. The `in`-node list is given as reference and changed in place.

In the latter case, we have to consider several possibilities. If the first instruction is a `Stop`, then it is simply discarded. If it is a sequence not beginning with an `out`-instruction, the node is simply reduced by adding the result of calling `threads` on the second part of the sequence to the result. If it is neither an `out` nor one of the aforementioned cases, the result of `threads` on the whole protocol of the node is added to the result. `threads` called on a protocol computes all pathes introduced by it.

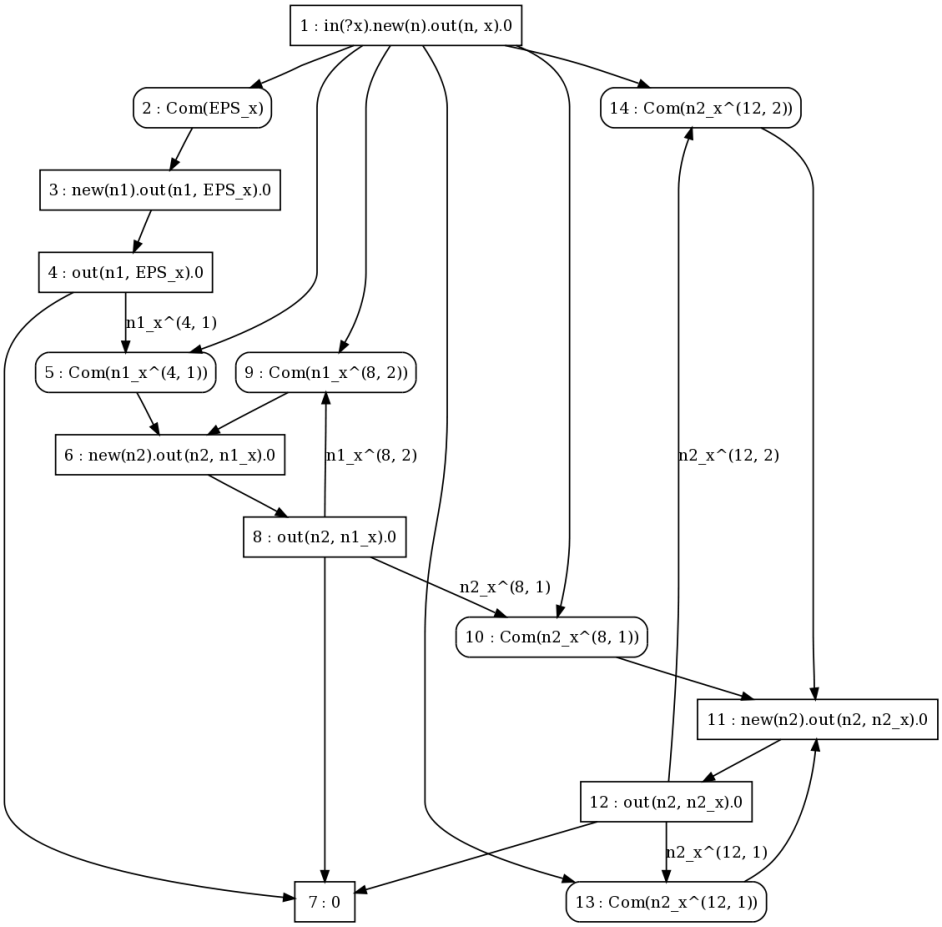


Figure 3.2: α-renaming in loops

Whenever we reach a sequence starting with an `out`-instruction new information becomes available to the environment, because everything sent enters its knowledge. This reflects the Dolev-Yao model where the attacker can listen on any channel. The method that adds something to the environment tries to break it down to the smallest possible term, e.g. by breaking cipher or splitting tuples. In addition the newly added terms are used to possibly break every cipher learned so far. This ensured that the environment contains everything that could be deduced by an attacker as well as the honest communication. In practice we make a distinction between primitives the environment cannot break, like ciphertexts where the key is missing, and atomic parts like messages or names. The first are stored in a separate list, which makes the matching later on easier.

This realization of the environment is a nice implementational trick. Due to the fact that it contains the honest communication as well as everything deduceable, the reduction yields the intended result as well as every possible communication.

```
let step in_nodes level = List.fold_left (fun a b ->
    if is_in_node b then
        (in_nodes := b :: !in_nodes;
         (reduce_in b) @ a)
     else
         match b#get_protocol with
             Sequence(Out(t), p) ->
                 (insert_E_env (t2gt (output t (b#get_id))) (b#get_id);
                   reduce_ins_with_out in_nodes b @
                   create_nodes b (threads p) @ a)
             | Sequence(_, p) -> create_nodes b (threads p) @ a
             | Stop -> a
             | _ -> create_nodes b (threads b#get_protocol) @ a
    )
    [] level
```

Figure 3.3: The step function

Reducing In-nodes

The actual reduction of `in`-nodes is identical, whether it was caused by new knowledge entering the environment or the `in` itself. First, all terms matching the structure of the `in`-instruction are computed. A structural match in this sense means every term that fits exactly the datastructure of the term expected by `in` in Caspa, e.g. identity A matches identity B but identity A does not match name m. The term inside the `in`-node and all possible matches are then passed to the `bind_abs` procedure, which is a faithful implementation of $bind_{abs}$ from [3]. Here it is checked, whether the two terms really match and if so, the integer components and a replacement function are returned. The replacement function represents the actual binding from variables to names. As clarification, the application of `bind_abs` on the two terms `in(x)` and `out(n)` results in a function that replaces `x` by `n`. In the main loop the according communication-nodes, annotated with the integer components, are added, linking together the `in`-node, the corresponding `out`-nodes and the reduced node. The reduced nodes are added as result to the new level-list. The communication-nodes never enter the level as they are only auxiliary constructs we utilize during trace generation.

Thread Coloring

As mentioned above we extended the analysis by features that improve the usability of the output of Caspa. One example is the optional coloring of different threads in the graph. Optional in the sense that the user can choose which graphs should be generated. The section about the Interface explains this further.

During the graph generation every node that is created gets a color assigned. The first node in the graph is neutral, i.e., white, and every node created inherits the color of his parent. As soon as a `parallel`-instruction is reached, each thread executed in parallel gets assigned a color. This allows for quickly distinguishing the different threads even in huge graphs and proved to be a great help when analyzing them by hand. Currently we chose six different colors because the number of visually easily distinguishable colors is restricted and even if a protocol has more than six threads alternating but repeating colors still improve the

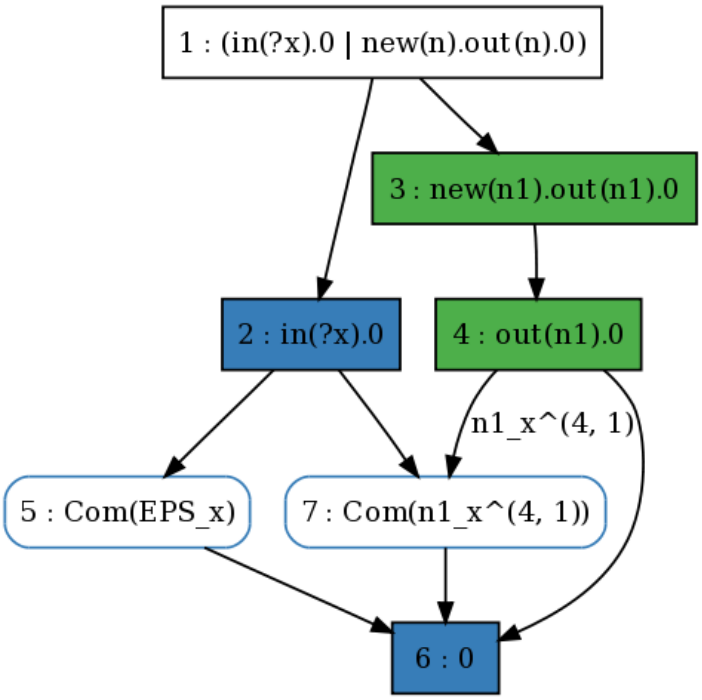


Figure 3.4: Example from Fig. 2.2 with colored threads

readability.

On the implementational side we utilized the `create_node` function depicted in Fig. 3.5 that actually creates the node object and inserts it into the graph. If the call to `threads` for the current node returned a list with length greater than one, the `create_node` method is given a flag that indicates that it should use new colors.

Fig. 3.4 shows the example from 2.2 with different threads colored. Note that communication nodes, depicted with rounded corners, can not get a color assigned because of the current Graphviz implementation which does not allow for specially shaped nodes together with colored filling. To address this, they get a colored border indicating to which thread they belong. We consider them belonging to the thread the corresponding `in`-node originates from.

Secrecy Tracking

Another interesting feature added is the secrecy tracking. So far we could only identify whether a name got public by entering the knowledge of the environment or not. However a more useful information would be to know where the information became public and why. To keep track of this information we added a hashtable that stores names as keys and a list of node-ids. Whenever new information is added to the environment during graph generation, this hashtable is updated too. There are two cases to consider. For a single name sent out plain, this is a simple mapping of the name that got public to the id of the node that sent it out. The second case occurs whenever the environment can break a ciphertext with the newly added knowledge or a newly added ciphertext can be broken with information already present.

```
let create_node parent prot' newcolor =
    let prot = get_alpharenamed prot' in
        try
        (
            check_restricted prot;
            let x = findnode !nodelist prot in
            (x#add_parent parent; None)
        )
        with Not_found -> let x = new node prot true in
            (Printf.printf "created new node with protocol: ";
             print_prot_chan prot stdout;
             Printf.printf "\n%!";
             nodelist := x :: !nodelist;
             x#add_parent parent;
             if newcolor then
                x#set_fillcolor (get_next_color ())
             else
                x#set_fillcolor (parent#get_fillcolor ());
             Some(x)
            )
```

Figure 3.5: The create_node function

In this case the names inside the ciphertext are all inserted separately into the tracking table together with a list of node-ids, consisting of the node sending the broken ciphertext and the id of all nodes that are required to break it.

To clarify the technique, consider the following example: as part of a protocol we have two processes, `out({ m | n })`, i.e., `m` is sent out encrypted with `n`, and `out(n)`. When the second process is transformed into a node in the graph, `n` enters the environment. Let this node have the id `y` and the node for the first process the id `x`. The tracking table is extended by an entry `n -> [y]`. As soon as the ciphertext from the first process is sent, the environment can use `n` to break it and thus `m` enters it too and the tracking table is extended by `m -> [x, y]`. `x` as `m` was sent there and `y` because there the information got known that was needed to brake the ciphertext.

Due to the fact that the code for secrecy tracking is added to the code that adds knowledge to the environment, building the graph is sufficient to collect all secrecy information for all names in the protocol. But on the other hand, not every name in a protocol has to be kept secret. It is quite reasonable that some names are sent out in plain. To filter, which names should be kept secret, we extended the protocol syntax to allow the protocol designer to define the names he expects to be secret. Section 4.2 explains the syntax of the file format for Caspa in detail.

Checking the secrecy property now is a straight forward task. We request the list of names that should be secret from the parser and then perform a lookup for each of them in the tracking table. If it is not contained therein, it is kept secret, but if the lookup succeeds we not only know that the name got public, we also get a list of nodes that were involved in the release of it. We further utilize this by offering a separate version of the graph, called secrecy graph, that highlights all such nodes in red.

Example - We will demonstrate this feature using the protocol introduced in Fig. 2.1,

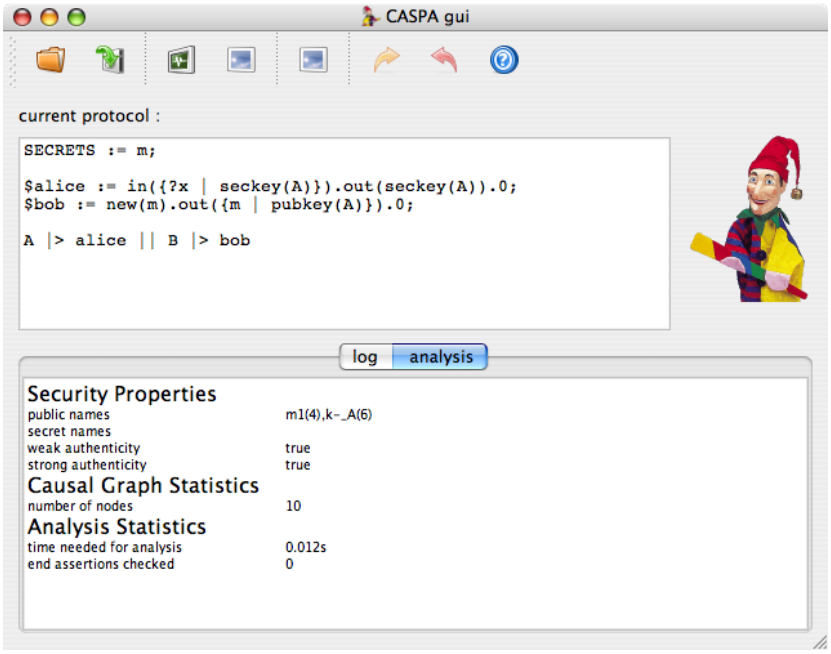


Figure 3.6: Protocol 2.1 in simplified form

which provides secrecy and authenticity of the exchanged message between two parties. As we are only concerned about secrecy in this example, we drop the assertions. Further we omit the identity and the nonce for improved readability. The remainder, depicted in Fig. 3.6, constitutes a protocol where `B` sends a message `m` encrypted to `A`. However, this protocol still preserves the secrecy of `m`. To introduce a weakness we send out the secret key of `A` after receiving the encrypted message.

As expected, Caspa identifies the name `m` as being public. The secrecy graph, shown in Fig. 3.7, highlights the output of the ciphertext containing `m` and the output of the secret key of `A`, since the latter is needed to decrypt the former.

3.2 Trace Generation

The second core algorithm in Caspa is the generation of traces and trace sets. These are needed to prove authenticity and cycle invariance.

Before beginning with the description of the algorithm we would like to clarify some expressions we will use. A trace is a single path in the graph from an arbitrary node to the top node and consists of the actions corresponding to the nodes along this path. Such a trace is denoted top down, so it starts with the top node and ends one node before the one we compute the traces for. A path is a set of traces that contains every trace that is causally needed to reach a given node. Whenever we talk about the traces or paths for a given node, we would like to compute all paths, i.e., all possible protocol sessions, that lead to this node or state. When we talk about structural paths we refer to intra-thread communication and communication paths refers to inter-thread communication.

On the implementational side a trace is an action list, a path a list of traces and the paths

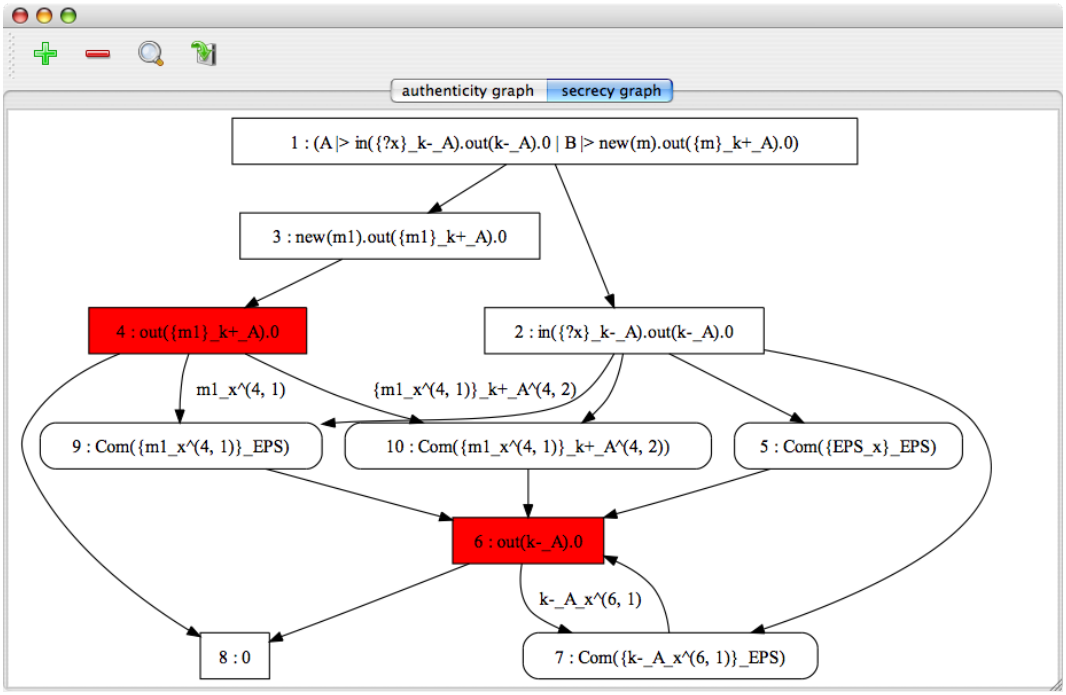


Figure 3.7: Secrecy graph for the protocol 3.6

```
let paths_table : (int, (prot_edge list * paths) list) Hashtbl.t = Hashtbl.create 100
```

Figure 3.8: Dynamic programming table definition

for a given node a path list, i.e., an `action list list list`. Additionally we decided to store a reference to the node, an action originates from, together with the action, so in fact we have `(action * node) list list list` as the resulting type. We will make use of this annotation for attack reconstruction during the authenticity check. Internally the algorithm uses type definitions for traces and paths for convenience.

The implementation we present here follows the definition from [3] directly, extended with dynamic programming techniques.

The algorithm and its functions are located in the `tracepaperdp` module. The main method is `paths` which invokes `paths'` with the given node as argument and an initially empty set of edges and afterwards postprocesses the result returned by `paths'`.

Whereas the idea of paths is clear to us, edges are new. Due to the representation of the graph as objects with references to each other there exist no explicit edges in Caspa. For the trace finding algorithm however it is important to use every edge leading to an `in`-instruction only once for a trace to ensure that every cycle is only passed once, so we store the edges used so far together with the resulting paths. For a single node we can thus have several different paths-edges pairs. In practice this can occur for example if we reach a node from different child nodes that of course use different edges to reach it. So a possible cycle may have already been used for the one node whereas the other one still may use it.

This explains why we invoke `paths'` with an initially empty edge list: no edges have been

```
1   let run =
2       if Hashtbl.mem paths_table n#get_id then
3       (
4           let possible_paths = Hashtbl.find paths_table n#get_id in
5           List.fold_left (fun a (be, bp) ->
6               if list_compare edge_compare be e = 0 then
7                   Some bp
8               else
9                   a) None possible_paths
10      ) else
11      (
12          None
13      ) in
14  match run with
15      Some x -> x
16      | None -> ...
```

Figure 3.9: Dynamic programming lookup

used so far. The first step of the algorithm is a lookup in the dynamic programming table for the current node (Fig. 3.9). If an entry was found we have to compare all the paths-edges pairs stored with the current edge list. If there is a complete match, i.e., the edges used are equal, the paths stored are the result for the whole function. If not, we have to compute the paths for the current node and add them to the table.

Before we go into detail with the algorithm let us have a closer look at the dynamic programming extension. In order to save results for later reuse we introduced a hashtable whose definition you can find in Fig. 3.8. It stores the id of a node as key together with a list of edges and paths pairs.

Before computing anything related to paths, Caspa computes the traces for all nodes in the graph top down and stores the results so that they can be reused. Here the unique ids assigned to nodes by the graph generation algorithm are very helpful. Since we create the graph in a breadth first way from top to bottom, in general a lower id for a node means that it is further up in the graph, i.e., it was created earlier. So we generate the traces for the nodes in the order of their id. For every node there is a high probability that we already computed the traces for the parents of this node.

After this computation every call to trace generation is a lookup.

As first step of the computation all edges that lead to the node we are considering are reconstructed using the references to the parents stored in every node object and the communication nodes introduced during graph generation. The latter are necessary because they store the message sent and the integer components from the corresponding `out`-nodes that are needed to reconstruct both types of communication edges, i.e., `ComOut`- and `ComIn`-edges. The former contain the `out`-node, the message sent, the integer component and the target node and the latter the `in`-node, the message sent and the target node. Structural edges connect nodes introduced by structural reduction and contain only the two nodes.

If the edge list is empty, the current node has no parents, so the paths for this node are empty. This marks the end of the recursion, an empty entry for this node together with the edges used is added to the hashtable and the empty path is returned. If the edge list is not

empty, we have to make a distinction between structural paths and communication paths referring to intra-thread communication and inter-thread communication respectively. To achieve this we split the edges computed into structural edges and communication edges and provide them as arguments to subfunctions called `communication_paths_of_n` (Fig. 3.11) and `structural_paths_of_n` (Fig. 3.10) along with the current edge list. The result is the concatenation of the results returned by these subfunctions. Again this result together with the current edgelist is added to the hashtable before being returned.

```
1 and structural_paths_of_n struct_edges e =
2     let pre_paths = List.fold_left (fun a b -> let pP = get_p_from_edge b in
3             action_to_paths (make_action_node pP#get_protocol, pP) (paths' pP e) @ a)
4         []
5         struct_edges in
6     pre_paths
```

Figure 3.10: structural_paths_of_n

`Structural_paths_of_n` iterates over the list of edges it receives as argument, extracts the corresponding action and adds it to every trace in every path returned by a recursive call to the `paths'` function with the source node of the current edge as initial argument.

`Communication_paths_of_n` is slightly more complicated. First the list of edges provided as argument is split into the one edge originating from the `in`-node and edges originating from an `out`-node. The latter list may be empty if this communication only contains messages created by the environment. Then it is checked whether the `in`-edge has already been used by doing a lookup on the list `e` of already seen edges provided as argument. The result is stored. Afterwards the message `m` expected by the `in`-node is matched against the message sent, extracted from the communication edge using `bind_abs` which is a faithful implementation of the $bind_abs$ procedure from [3]. If either the matching does not bind, which is basically only a sanity check, or the `in`-edge has already been used, which was determined and stored beforehand, the computation ends and the function returns an empty list. Otherwise the `in`-process `inP` and the `out`-processes from the `out`-edges are extracted. The next step is the computation of the paths for `inP` and all `outP` by a recursive call to the `paths'` function, like we did it for structural edges, but this time the `in`-edge is added to the list of already used edges when invoking `paths'` to ensure that every `in`-edge is only used once by every path. To all paths in the result of the recursive call an `out`-action containing the message sent from the `ComOut`-edge and the corresponding `out`-node is added as well as an `aincom`-action with the integer-component and the `com`-node. The result returned by `communication_paths_of_n` is obtained by adding the paths for the `in`-node to all paths in the big set computed for the `out`-nodes.

3.3 Authenticity Checking

Before we explain the implementation of authenticity checking in Caspa we shortly repeat what we want to prove. A graph provides authenticity if and only if for every `end`-node there exists at least one trace in every path for this `end`-node with a matching `begin`. A matching `begin` is a `begin`-assertion where the indices, the annotated names crossed over and the session messages match the respective ones from the `end`-assertion. Additionally the

```
and communication_paths_of_n com_edges e =
    let in_edge, out_edges = split_in_out_edges com_edges in
    let in_edge_in_e = List.fold_left (fun a b -> a ||
            edge_compare_eq_ComIn b in_edge)
        false
        e
    in
    let m = get_m_from_comin_edge in_edge in
    let sigma, bound = match bind_abs m (labelerazor_term
            (get_sent_from_com_edge in_edge)) with
        Some (_, f) -> (f, true)
        | None -> raise (Invalid_argument "Communication_paths_of_n:
            inconsistent edge found (tracepaperdb.ml)")
            (*((fun z -> z), false)*)
    in
    if ((not in_edge_in_e) && bound) then
        (
        let inP = get_p_from_edge in_edge in
        let m_sigma = replace_term sigma m in
        let in_S = action_to_paths (AIn(m_sigma, m_sigma), inP) (paths' (inP)
            (in_edge :: e))
        in
        let big_path_list =
            List.fold_left (fun a b ->
                let gi' = get_ic_from_comout_edge b in
                let gi = get_gi_from_comout_edge b in
                let outP = get_p_from_edge b in
                let out_S = paths' outP (in_edge :: e) in
                    action_to_paths (AIncom (gt2t gi', m_sigma),
                        get_comnode_from_com_edge b)
                        (action_to_paths (AOut(t2gt gi), outP) out_S) :: a)
                []
                out_edges
        in
        combine_paths in_S big_path_list
        )
    else
        []
```

Figure 3.11: communication_paths_of_n

messages must be from the same session. So in order to prove authenticity for a given graph we have to compute the traces for every `end`-assertion.

The algorithm implementing authenticity checking follows the intuition by first computing a list of `end`-nodes from the graph. Then for every identified candidate it is checked whether it provides authenticity and if so, whether it provides weak or strong authenticity. To return as much information as possible the algorithm keeps track of all `end`-assertions providing authenticity, weak authenticity and no authenticity separately. The module provides access functions to retrieve the corresponding lists and information like the number of assertions checked. In addition every node that does not provide authenticity is marked by setting its

fillstyle attribute to "dashed" resulting in the node being rendered differently when the graph is displayed in the graphical interface or rendered using Graphviz.

Individual `end`-assertions are checked by the `check_candidate` method that computes the sets for its argument and then checks every set. In addition the first set that does not provide authenticity is marked. The individual sets are checked by the `check_set` method that again only iterates over all traces in a given set and invokes `check_trace` on them.

A single trace provides authenticity if for every `end` in this trace there is a matching `begin`. This is computed by extracting all `end`s and `begin`s together with their positional index from the trace and then checking the matching for every `end` against the `begin`s. A `begin` matches an `end` if its positional index in the trace is smaller than the one of the `end`, the indices annotated in the assertions match, the identities match crossed over and finally the session messages for the messages send and the nonce match and originate from the same session.

Example - We revisit the protocol 2.1, i.e., a message exchange between two parties. The analysis using Caspa concludes that the protocol provides strong authenticity.

As explained previously, in order to prove authenticity, Caspa has to check the traces for all `end`-assertions in the graph. For our protocol, there is only one `end` with seven trace sets. This makes it feasible to follow them manually. Fig. 3.12 shows one of the seven sets. We can easily check that there is at least one trace with exactly one matching `begin` in it, i.e., $begin^1_{n1_{(x)}}\left(A, B, m1_{(z)}\right)$, but none with two matching `begin`s. The complete trace sets can be found in appendix A.1.

Paths of $end^{1}_{n1}(A, B, m1)$

$$
\left\{
\begin{array}{l}
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1^{8,2}_{(x)}, m1^{8,3}_{(z)}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin^{1}_{n1_{(x)}}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}_{(x)}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1^{8,2}_{(x)}, m1^{8,3}_{(z)}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin^{1}_{n1_{(x)}}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}_{(x)}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}_{(z)}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}_{(x)}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin^{1}_{n1_{(x)}}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}_{(x)}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\
\quad :: begin^{1}_{n1_{(x)}}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}_{(z)}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin^{1}_{n1_{(x)}}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}_{(z)}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin^{1}_{n1_{(x)}}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right)
\end{array}
\right\}
$$

Figure 3.12: One trace set from the paths of the `end`-assertion from example 2.1

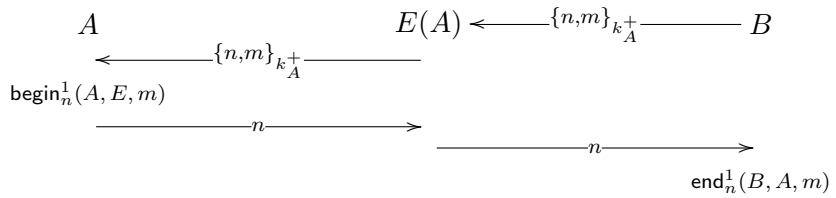
3.3.1 Attack Reconstruction

To provide a better intuition of a possible flaw in a protocol that does not provide authenticity we extended the code to mark the first set that fails in the `check_candidate` function. This is done by setting the color attribute of all nodes belonging to this set to `red`. This results in a highlighting of the whole communication that does not provide authenticity once the graph is rendered.

Although the restriction to the first set failing seems a bit arbitrary, it is important to notice that it is not useful to mark more than one set. We have discovered during tests that highlighting more than one set leads to a situation where a huge part of the graph is red and additionally it is unclear to which failing set a marked node belongs. Using different colors for every failing set would be unpractical too because the number of expressive colors is restricted and the problem arises how to color nodes that belong to more than one failing set. However if we only color one set, choosing the first one is a feasible solution in practise.

As a motivation for this extension remember that Caspa implements an abstract interpretation technique. If Caspa claims that a protocol does not provide authenticity, this only is a possible flaw and we have to analyze the protocol further using different techniques or by hand. But the traces for real life protocols fill several pages and the graphs usually are too big to fit on a screen in a readable size. The highlighting however helps greatly by providing a visual guideline to a failing path. This allows users familiar with the graph representation to easily track the communication and get insights into what is possibly going wrong.

Example - As an example we refer again to protocol 2.1. Since this protocol provides strong authenticity, we have to introduce a weakness. This is done by removing the identity in the ciphertext sent by the initiator. The modified protocol suffers from a man-in-the-middle attack:



Intuitively spoken, `B` believes he is talking to `A`, but in fact he is talking to the attacker. Note that, while being present in the graphical protocol description, assertions involving an attacker do not need to be modelled in Caspa as we do not wish to check if an attacker can authenticate. For simplicity we omit them (see Fig. 3.13).

The analysis of this modified version of the protocol reveals that our change indeed broke authenticity. The secrecy graph, depicted in Fig. 3.14, shows that in the highlighted session `A` was never willing to authenticate with `B`, although he authenticates `A` in the end. This constitutes an attack.

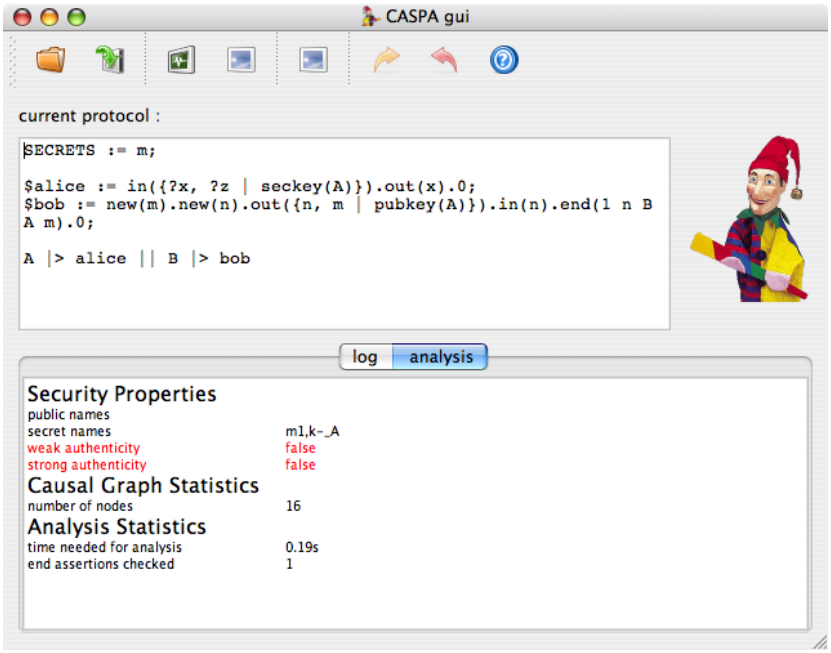


Figure 3.13: Protocol 2.1 modified to not provide authenticity

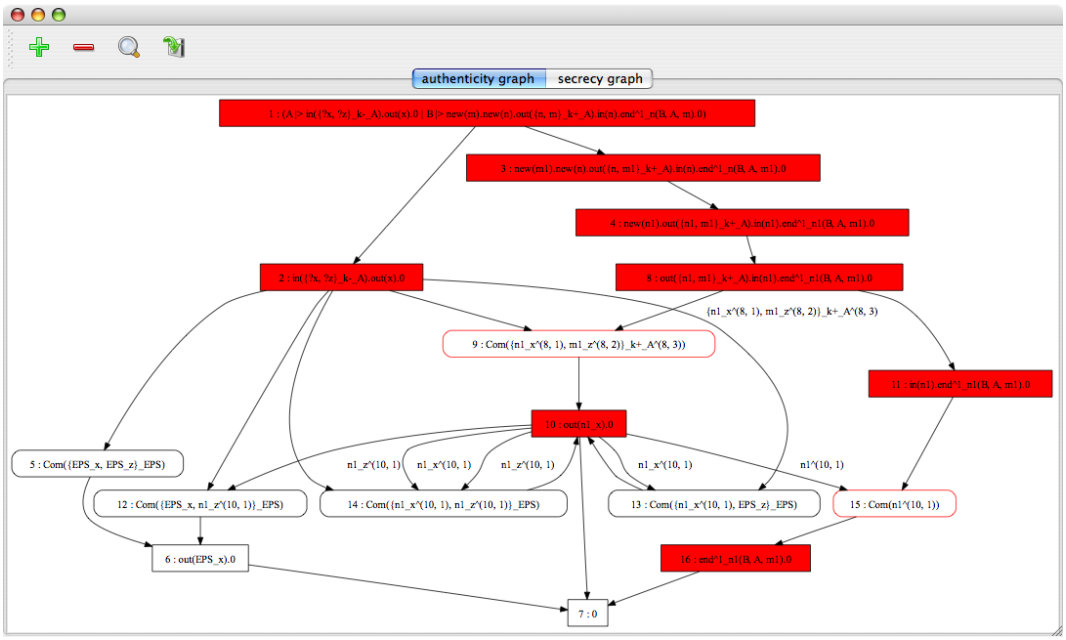


Figure 3.14: Authenticity graph for the protocol 3.13

Section 4

Interface

Caspa offers various ways for the user to interact with it, beginning with the input files Caspa accepts over the command line switches in the console version of it to the easy to use gui. Both versions of Caspa work on the same code and offer the same possibilities, but while the command line version offers a fine granular selection of the output for the expert user, the graphical interface reveals the full power of Caspa with a single mouse click. In the following we will present the individual features of the command line and the graphical version of Caspa.

4.1 Command Line Program

On the command line Caspa can be invoked with several switches that determine how much output is produced, which graphs are generated and for which node the paths are outputted. The general usage string, depicted in Fig. 4.1, follows the standard unix conventions:

An overview of all switches and a short explanation can be found in table 4.1. The protocolfile must always be provided because this is the inputfile with the protocol to be checked. All other switches are optional but some of them may require others to be set to work correctly. When invoking Caspa with a wrong switch or with `-help` a short overview is displayed on the console giving the short explanations you find in table 4.1. In the following we will explain them in more detail.

The `-s` flag allows the user to define for which node in the graph the traces should be generated and pretty printed into a file called `protocol.tex`, where `protocol` is the basename of the inputfile Caspa was invoked with. The ids of the nodes are displayed in the graph, so this option is only useful after seeing the graph, in a second run for example. In this case it offers the possibility to print out the traces and trace sets for any given node in a readable form. This does not improve the automated analysis, but allows the user to check the traces manually by compiling the tex output to a pdf or a postscript document, or to use it further. Fig. 2.3 for example is the output of the execution of Caspa on the protocol from Table 2.2

```
./main [-s <startid>] [-T outputtype] [-t type] [-c] [-v level] [-p acds]
                                <protocolfile>
```

Figure 4.1: Usage string for Caspa

-s	the id of the node to start the traces from (as default the first node in the graph (id 1) is taken)
-T	the type of output to render
-t	the trace function to use (1: paper version with DP, 2: paper version without DP, 3: old version)
-v	verbosity level, 0 by default
-c	if set, using slow algorithm for cycleinvariance check
-p	the print modifiers determine which images will be generated. a for authenticity, s for secrecy, c for colored and d for the default graph. -p ac for example would generate two graphs, one for authenticity and one colored
-help	display this list of options

Table 4.1: Switch table for Caspa

with `-s 6`.

The `-T` switch allows the user to define what type the images of the graphs should have. By default Caspa produces a .dot file of the graph that can be displayed using Graphviz and no traces. Using the `-T` flag it is possible to produce an image of the graph directly, without the need of calling Graphviz manually. The type of output possible is defined by the Graphviz implementation on the system running Caspa. Typical examples are `-T png`, for png images, or `-T jpg`, for jpgs. To see a full list of options supported use the command `dot -T?`. If `-T` is used, Caspa will create a temporary file, print the dot output in there, invoke `dot` program from Graphviz on it and produce the output specified by the user. The resulting file will then be called `protocol.type`, e.g. `rsa.png`. The temporary file will be deleted afterwards.

Using the `-c` flag the user can chose to use a second algorithm for checking cycle-invariance. The one used by default is much faster than the slower one, which was implemented earlier. The faster technique can be proven equivalent and thus is the default.

The `-p` flag results from the different extensions to Caspa introduced in the section about the architecture, i.e., attack reconstruction and colored threads. Given this option the user can chose which graphs he wants to be created. The flag expects a string as argument which may consist of an arbitrary combination of the letters `a`, `c`, `d`, and `s`. The individual letters are interpreted, where `c` stands for a graph with colored threads, `s` for a secrecy graph, where the nodes where names enter the knowledge of the environment are highlighted, `a` for the authenticity graph and `d` for the default graph without any color.

Last, `-v` allows for defining the level of verbosity for Caspa. Every output statement in the code has a number attached defining the level of verbosity it has. The higher the number the more advanced this output is. By default a level of 0 is assumed so only the default outputs are printed. When using `-v 4` for example, every (debug-) output is shown. The logging mechanism used in Caspa is further explained in 4.1.1.

4.1.1 Logging

In addition to the images of the graph and the traces Caspa can also produce a lot of textual output on the console during execution. This output reflects the current computational steps executed and their respective results. Even for reasonable small protocols the amount of

output generated is huge. Therefore it is usually hidden. However, on demand it can be activated using the level of verbosity option `-v` from the command line client. To make this possible we implemented a powerful logging method that clearly separates data and representation on output.

Prior to implementing an own system we tried to extend the `printf` function of Ocaml with specifiers for our data structures. `printf` however is a highly complex function because its type changes according to the modifiers and the number of arguments. This requires external calls to a non-Ocaml library deep inside the implementation of Ocaml itself. At this point we decided for a pure functional implementation that works without external calls or additional libraries. This had the downfall, that we do not create a type that fits the arguments of the actual call, but have to create a function with one type that fits all possible calls.

A function that should be able to output every part of the protocol syntax separately however needs as many arguments as there are different types. This would make such a function inconvenient to use because we had to always provide all arguments. The solution is given by Ocaml itself. Ocaml allows the use of optional arguments. These arguments can be provided, but do not have to and can be assigned a default value. In the final version the `log` method prototype optionally accepts a list of arguments for every type. The only argument that is non-optional is the format string.

The core part of this system is the `logmanager` that administrates all loggers registered for the current application and processes every log statement in the code. Every logger is registered together with a verbosity level. All messages with a verbosity level smaller or equal to the one of the logger are passed to it by the manager. The `-v` option allows the user to set the loglevel for the default `logger` that is created.

Along with the logmanager we implemented a virtual base class `logger` that every logger has to inherit. This ensures that the logger is compatible to our system. The `log` method every logger has to implement expects one argument list for every datatype we introduce in Caspa and a format string, similar to the one `printf` uses. It may contain specifiers similar to `printf`, but much simpler, namely a single letter preceded by a `$` for every type. The order the specifiers appear in is the order the arguments are plugged in, e.g. the first protocol specifier will be replaced with a representation for the first element of the `?protocols` argument. Any mismatch between specifiers and number or type of arguments is considered an error. While the default logger outputs text on the console, possible new loggers could output into a file directly, on the screen using a window system or further process the information.

This flexibility of course cannot be addressed by the user directly, but is provided for experienced programmers that intend to modify or extend Caspa to their needs. Inside the Caspa modules we avoided `printf` completely and used the `log` statements instead.

4.2 Input File Format

Protocols given as input to Caspa, on the command line as well as in the graphical interface, are written in a powerful syntax, the prot file format, which is an extension of the grammar shown in Fig. 2.1. For practical reasons we allow comments in protocol files. Everything behind the letter `#` is considered a comment and is ignored during parsing. However it is possible to access the comments of the most recently parsed file. They are output for example on the console after a protocol was parsed successfully. In the following we will explain the

extensions to the grammar.

First of all we allowed the definition of sub-protocols and assignment of identifiers to them. This makes the definition of protocols easier and more readable. Figure 4.2 shows the basic example from 1.1 in the prot format accepted by Caspa. The threads for Alice and Bob are defined separately and assigned to the identifiers that are used in the final protocol definition on line 17.

```
1  # This is a simple challenge response protocol:
2  #
3  #     A                   B
4  #       <- {B,n,m}k+A --
5  #   begin 1 n (A,B,m)
6  #         ----- n ------->
7  #                     end 1 n (B,A,m)
8  #
9  # provides strong authenticity
10 # the message m is secret
11
12 SECRETS := m;
13
14 $alice := in({B, ?x, ?z | seckey(A)}).begin(1 x A B z).out(x).0;
15 $bob := new(m).new(n).out({B, n, m | pubkey(A)}).in(n).end(1 n B A m).0;
16
17 A |> alice || B |> bob
```

Figure 4.2: Protocol in prot format

Another extension we see in this example is the definition of secret names. In a protocol some of the names may intentionally be sent in plain over the network without losing security of the protocol itself. The `SECRETS` section in the input file offers the designer the possibility to manually define which messages should be kept secret in a protocol. Only the messages defined here will be checked for secrecy by Caspa.

The parser itself is realized using the Ocaml variants of lexx and yacc. The parser source file `parser.mly` contains a full definition of the grammar Caspa accepts and `parser.mll` the primitives that are recognized. Any error during parsing makes the command line program stop and the graphical interface output an error. The exact line number and position where the error occurred is output.

4.3 Graphical Interface

Maybe the most powerful aspect of Caspa in terms of usability is the graphical user interface we provide (see Fig. 4.3). The gui offers the user the possibility to easily load, modify and save protocols, to analyze them using one mouse click and viewing the different aspects of the graph at the same time.

After a prototype version written in lablTK, a TK/TCL extension for Ocaml, we decided to go for a more complex approach using Qt [11], a state of the art toolkit for graphical interfaces. Qt is platform independent and maybe the most powerful kit around. As Qt uses C++ [15] as programming language we had to find a way to combine the Caspa logic written

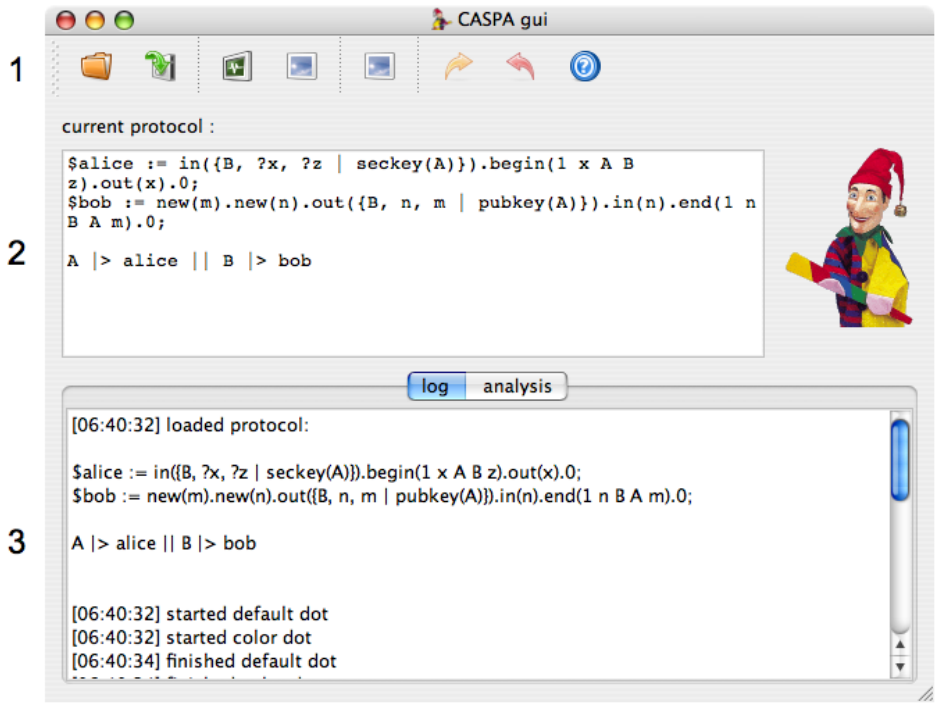


Figure 4.3: The Caspa Gui

in Ocaml and the interface code in C++.

The solution we chose was to create a static C library from the Caspa code. To achieve this goal we created a new Ocaml source file that encapsulates all function calls to Caspa and a wrapper written in C that defines the interface of the library. This is a first step towards a basic library for static protocol analysis. Other applications could use methods from the Caspa library for their implementation without being bound to Ocaml as programming language. Another benefit from this solution is the strict separation of model and graphical representation. Up to now only two functions are exported. One that loads a protocol from a file, and one that analyzes this protocol. For our graphical interface this is sufficient, for a general purpose library more fine grained functionality should be exported.

A closer look at the mentioned wrapper shows that it not only defines the interface but also transforms exceptions possibly thrown in the Ocaml code into C++ exceptions passed to the application using the library. In our graphical interface we create a small error dialog informing the user about the cause of the exception. Figure 4.4 shows the code of the wrapper. It defines two C functions that expect a string as argument and return a string as result. Both register on load one closure function and ensure that this costly process is only done once. The return value is then a callback to the Caml library. In the result field it is checked whether we have an exception.

The leading idea in the design of the Caspa gui was to reveal the full power of our analysis technique with only a few functions. To achieve this goal we separated the protocol analysis process into two major parts, the parsing of a protocol together with graph generation, and the analysis itself. While parsing usually only takes a few seconds, the analysis can be very time consuming. By this separation we let the user decide when the time consuming process,

```
1  extern "C" {
2  #include <string.h>
3  #include <caml/mlvalues.h>
4  #include <caml/callback.h>
5  #include <caml/alloc.h>
6  }
7
8  char* load_protocol(char* filename)
9  {
10     static value* load_protocol_closure = NULL;
11     if(load_protocol_closure == NULL)
12         load_protocol_closure = caml_named_value("load_protocol");
13     value v = caml_callback_exn(*load_protocol_closure, caml_copy_string(filename));
14     if(Is_exception_result(v))
15         throw strdup(String_val(Field(Extract_exception(v), 1)));
16     else
17         return strdup(String_val(v));
18 }
19
20 char* analyse_protocol(char* filename)
21 {
22     static value* analyse_protocol_closure = NULL;
23     if(analyse_protocol_closure == NULL)
24         analyse_protocol_closure = caml_named_value("analyse_protocol");
25     value v = caml_callback_exn(*analyse_protocol_closure, caml_copy_string(filename));
26     if(Is_exception_result(v))
27         throw strdup(String_val(Field(Extract_exception(v), 1)));
28     else
29         return strdup(String_val(v));
30 }
```

Figure 4.4: wrapper code in C

that cannot be interrupted and is blocking at the moment, starts.

In addition we source the generation of images out to own threads that are controlled by the main application. This reduces the time Caspa is busy to the computational time of our functions. External processes like the call to Graphviz for generating images do not influence the gui and can be interrupted, e.g. by a new parse command.

To further improve the usability of Caspa we only activate possible options at any given time of execution. This is realized by a small internal state machine the gui maintains. Every action from a user leads to a state transition and the new state determines the possible options. Qt allows us to activate or deactivate buttons. So a state transition results in a set of activations and deactivations. Due to the fact that the menu entries in the main menu and the toolbuttons are internally the same objects we only have to activate or deactivate the actions once and get a consistent state as result.

In general the Caspa gui offers two ways of providing protocols to be analyzed. The first is to load a protocol from a file, similar to the command line interface. The second option is to provide a protocol "on the fly" using the protocol editor and then parse it directly. This allows a user to modify a loaded protocol or to create a protocol from scratch, parse it, check

the result, modify it again, i.e., an interactive workcycle.

4.3.1 Mainwindow

The main window of the graphical interface, Fig. 4.3, consists of a toolbar at the top, the protocol editor in the mid-section and an information field in the lower half of the window. The toolbar, depicted in Fig. 4.5, offers a button for every functionality Caspa offers. From left to right this is loading (a) and saving (b) protocols, analysis (c), analysis graph (d), protocol graph (e), parsing (f), undo (g), and finally help (h). All these functions are available from the menu bar too using the same icons and descriptions.

Figure 4.5: The Toolbar

Saving (b) and loading (a) protocols work as expected. A click on the respective toolbutton opens a dialog that lets the user browse the file system and specify a file to save to or load a protocol from. As soon as a protocol is loaded, i.e., it is parsed and the graph built, the show graph button becomes available. This may take a few seconds because in the background we still use Graphviz to transform the graph into an image.

A click on the "show graph" buttons, namely (d) and (e) opens a new window, the graph display window, which is described in more detail in 4.3.2.

The analysis can be started using button (c) once a protocol was loaded or parsed successfully. This process may take some time depending on the number of communications in the protocol and cannot be interrupted. As soon as the analysis is finished the results are printed to the information window. Subsequent clicks on button (c) have no effect as long as the protocol is not changed. As mentioned above the rendering of the graph is started in separate processes. When they are ready the analysis graph button (d) becomes available.

The parse button (f) passes the current protocol in the protocol editor (2) to Caspa. Internally the current content of the editor is stored to a temporary file and this file is passed as argument to Caspa as if a user loaded the protocol. This allows us to re-use the function call for load protocol defined in the wrapper. If the parsing succeeds the protocol entered can be analyzed and its graph displayed, else the exception thrown by the parser is displayed as a parsing error.

As soon as the user modifies the protocol in the editor the undo button (g) gets available, but all analysis and display buttons get disabled. Undo restores the last loaded or parsed protocol, analysis results and graphs together with the gui state, e.g. whether the protocol has already been analyzed. The state then re-enables the appropriate buttons.

The help button (h) opens a help window that explains the general usage of Caspa and the grammar for the prot format. Fig. 4.6 depicts the online help.

The protocol editor (2) is a single text editing window that allows copy and paste. If the amount of space is not sufficient to display all lines in the window a scrollbar appears.

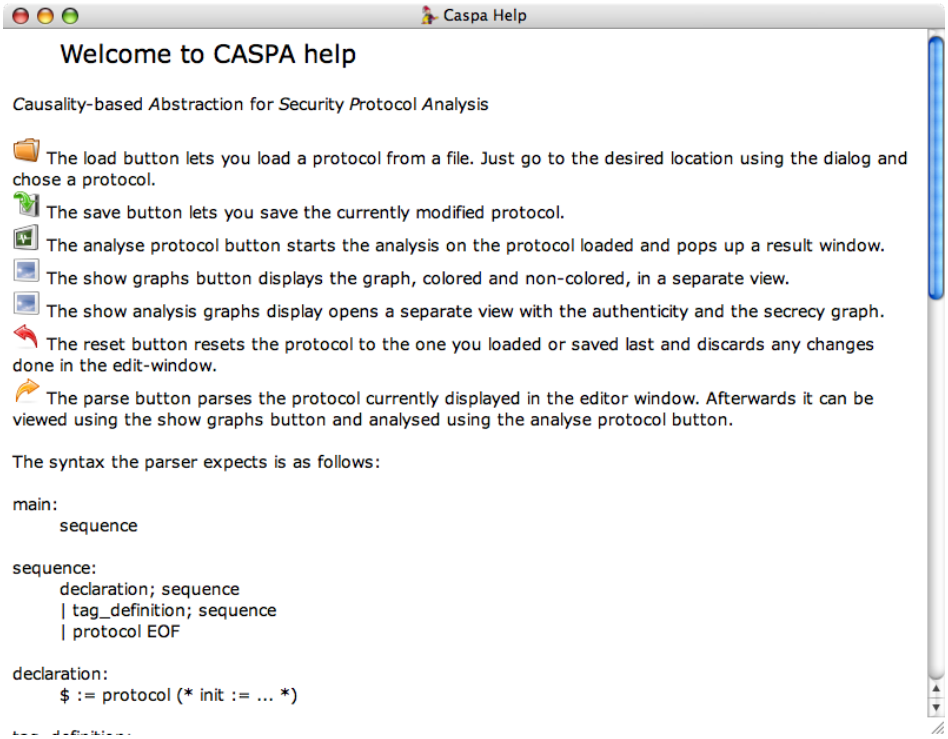


Figure 4.6: Online help

The information field (3) consists of two pages organized as tabs. Both pages are read only text fields that display information from Caspa. The one selected by default, the log window, shows process information like the name of the protocol loaded. The other one, the analysis result window, shows the analysis results, e.g. the number of nodes in the graph and the secret names. Both windows automatically add scrollbars if the content is too much to be displayed using the space available. Additionally both windows scroll automatically to the most recent information available.

4.3.2 Graphdisplay

The graphdisplay window, depicted in Fig. 4.7, is a viewer for the graphs generated by Caspa. It consists of a toolbar that offers basic controls at the top of the window and the view itself displaying the graph. One display window can contain several images. If so, they are organized as tabs and the user can switch between the displays using the tab control at the top of the view.

On the implementational side the interface of the display class consists of one function that allows the addition of images. The expected file format is SVG because this allows us to zoom in and out nearly arbitrarily. Whenever a file is added successfully, i.e., the filename was found and the image could be loaded, a new scene together with a new view for this scene is created and added as a new tab.

The toolbar offers four basic functions to the user. From left to right these are ”zoom in”, ”zoom out”, ”fit to size” and ”save”. The ”zoom” functions work as expected, they

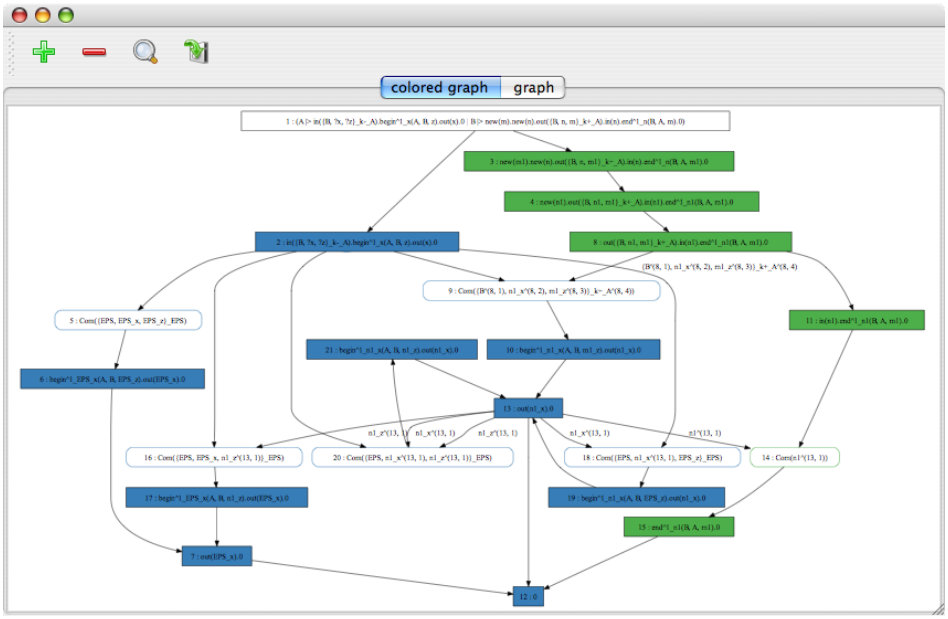


Figure 4.7: The graph display window

enlarge or shrink the currently displayed image. "Fit to size" scales the image to fit into the current window. This is especially interesting as it returns to an overview of the whole image from any zoom state. "Save" allows the user to save a copy of the image displayed. This feature is important for the Caspa gui because the images of the graph are only created temporarily to display them. If a user wants a permanent copy he can use this function from the graphdisplay. As soon as an image becomes too big to be displayed in the window scrollbars appear. Next to zooming the user can use the scrollbars for navigation or directly click on the point of interest using the mouse causing Caspa to center the scene on this point.

Section 5

Experimental Results

In oder to get an impression of the performance and to assess the utility of Caspa we ran it on a subset of the AVISPA library [2]. The results are reported in Table 5.1. Caspa succeeded in the analysis of safe protocols and it failed to establish security proofs of flawed protocols as expected. In comparison to TA4SP [12, 5], Caspa did not only improve in terms of performance, it is also capable of dealing with a larger set of protocols: the symbol - in the table means that the protocol is not supported by the according tool, while the symbol × denotes that the protocol guarantees only authenticity properties, which can be verified by Caspa and but not by TA4SP. For additional comparison we added the running times of OFMC [4], the most advanced model checker in the AVISPA tool, when the analysis is constrained to three executions. In this setting, OFMC performs significantly better. However, model-checking does not guarantee termination and asserts security guarantees for only a small number of executions.

Protocol	CASPA	TA4SP	OFMC	Protocol	CASPA	TA4SP	OFMC
CHAPv2	0,93s	10,59s	0,32s	NSPK	0,13s	7,56s	0,01s
CRAM-MD5	0,09s	-	0,71s	NSPK-KS	28m	-	1,1s
EKE	0,81s	7,56s	0,19s	NSPK-fix	0,08s	0,98s	0,18s
IKEv2-CHILD	0,31s	-	1,19s	NSPK-KS-fix	7m	-	24,86s
ISO1	0,05s	×	0,02s	SHARE	0,4s	14,38s	0,08s
ISO3	1,08s	×	0,04s	UMTS-AKA	0,04s	0,51s	0,02s
LPD-MSR	0,05s	-	0,02s	APOP	0,44s	×	2,94s
LPD-IMSR	0,37s	-	0,08s	DHCP-DA	1,03s	-	0,06

Table 5.1: Protocol results, conducted on a Pentium-IV 3GHz 1GB under linux

As preparation to the evaluation we had to identify the protocols Caspa is capable of analyzing and to translate them from the Intermediate Format protocol language [2] into the `prot` format. The former is important as up to now Caspa is only able to handle symmetric and asymmetric encryption, hashes, and signatures, however exponentiation for example is out of scope. To facilitate the latter task we developed a translator from the Intermediate Format to our dialect of the spi-calculus. The translation involves manual steps, such as specifying the owners of the keys and defining suitable correspondence assertions. These steps require basic familiarity with our language and understanding of the protocol.

Section 6

Conclusion and future Work

We have introduced Caspa, a fast and stable implementation of an static analysis technique for security protocols. We briefly motivated the approach and explained the underlying technique as well as the implementation of the key algorithms and their extensions and optimizations. Finally we presented all ways to interact with Caspa including the graphical user interface.

As mentioned above the code of Caspa is already revised and optimized. Further improvements of the runtime can only be achieved by applying implementational tricks, e.g. the abbreviation of loops using exceptions, or by changing the algorithms used. Yet the utility of Caspa still can be improved.

When deriving attacks, the current implementation only considers the first trace set failing. However there may be even more such sets, but to maintain readability it is impossible to mark more than one trace set in a single image. However the graphical interface already offers the possibility to display several graphs of the protocol analyzed. This has to be extended to enable the user to see all possible flaws. The first option is to create a single graph for every failing trace set or second, all fails are reported to the graphical interface and the gui offers the user a menu where he can select the graph he wants to inspect.

The first option requires some work on the code of Caspa itself. All trace sets failing have to be recorded and the corresponding images created. The graphical interface needs no change as the graphdisplay class already allows for the addition of an arbitrary number of images. The latter option however only requires small changes to Caspa, i.e., recording and exporting of trace sets not providing authenticity, but major changes on the representation of graphs in the graphical interface. In the current implementation graphs are images. But to be able to interactively highlight different aspects on the same graph it has to be a real data structure whose properties can be changed. This requires graph representation and layout procedures.

As a side effect a more complex representation of graphs would offer a new grade of flexibility and interactivity to the graphical interface. Nodes can be reordered, or more general, moved by the user, graphs re-arranged to point out different aspects and nodes selected using a mouse-click and corresponding information made available to the user. This extends to the protocol currently analyzed. A highlighting of a node in the graph and corresponding part of the input protocol in the protocol editor and vice versa becomes available, strengthening the utility of Caspa further. As an example consider a protocol where the name `n` becomes known although it should be kept secret. The secrecy graph already highlights all nodes that lead to this condition, but the enhanced version also directly highlights the corresponding part of

the input protocol in the editor, making identifying and correcting possible flaws easier. In general a better representation of graphs improves the utility of Caspa as an analysis tool for cryptographic protocols because it emphasizes the coherence between the protocol and causal relations in the graph.

A different aspect that can be enhanced is the connection of Caspa to other applications, e.g. the graphical interface. As explained earlier we chose an approach where we define an interface and compile Caspa to a static C library. The interface however only offers two very abstract functions that provide access to graph generation and analysis. It has to be extended to allow a more fine grained access to the methods in Caspa. This makes the library interesting for more applications than its own graphical interface because single algorithms become available that may be interesting for other projects as well. However this imposes some care when selecting the functionality that is exported. At the moment there is a lot of internal state in Caspa and part of the functions are not self-contained, i.e., there are functions which expect that other functions have been called beforehand. So exporting every function can lead to inconsistent program states which will cause erroneous results.

In addition it is useful to provide a dynamic library in addition to the static one. Whereas the latter contains the Ocaml bytecode in addition with the runtime files of Ocaml, the former needs Caspa to be compiled to native code using `ocamlopt`. Next to a better integration into the unix environment, Ocaml code compiled to native code is simply faster than the bytecode variant. During experiments we noted that the runtime of the native version of Caspa is faster than the bytecode variant by at least a factor of two. However, up to now the bytecode variant does not work together with the graphical interface. The reason for this is related to the linking of the dynamic library into the gui code written in C++ using the Qt toolkit. This has to be further analyzed before changing Caspa completely to bytecode.

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, L. Cuellar, P.H. Drielsma, P. 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. Computer Aided Verification'05 (CAV)*, LNCS, pages 281–285, 2005.

[3] Michael Backes, Agostino Cortesi, and Matteo Maffei. Causality-based abstraction of multiplicity in security protocols. In *Proceedings of 20th IEEE Computer Security Foundation Symposium (CSF)*, June 2007.

[4] D. A. Basin, S. Mödersheim, and L. Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[5] Y. Boichut and T. Genet. Feasible trace reconstruction for rewriting approximations. In *Term Rewriting and Applications (RTA 2006)*, pages 123–135, 2006.

[6] M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authenticity protocols. In *Proc. 13th European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004.

[7] M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication. *Journal of Computer Security*, 2007.

[8] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *Technical Report 39*, February 1989.

[9] D. Denning and G. Sacco. Timestamps in key distributed protocols. *Communication of the ACM*, 24(8):533–535, 1981.

[10] Laboratoir Spécification et Vérification. Common syntax. http://www.lsv.ens-cachan.fr/spore/format.html.

[11] Qt Software (formerly Trolltech). Qt - a cross-platform application and ui framework, 2008. URL: http://www.qtsoftware.com.

[12] T. Genet and V. Tong. Reachability analysis of term rewriting systems with timbuk. In *Proc. Artificial Intelligence on Logic for Programming (LPAR '01)*, pages 695–706. Springer-Verlag, 2001.

[13] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The objective caml language, 2002. URL: http://caml.inria.fr/.

[14] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[15] Bjarne Stroustrup. C++, 1983. URL: http://www.open-std.org/jtc1/sc22/wg21/.

Appendix A

Appendix

A.1 Additional Figures

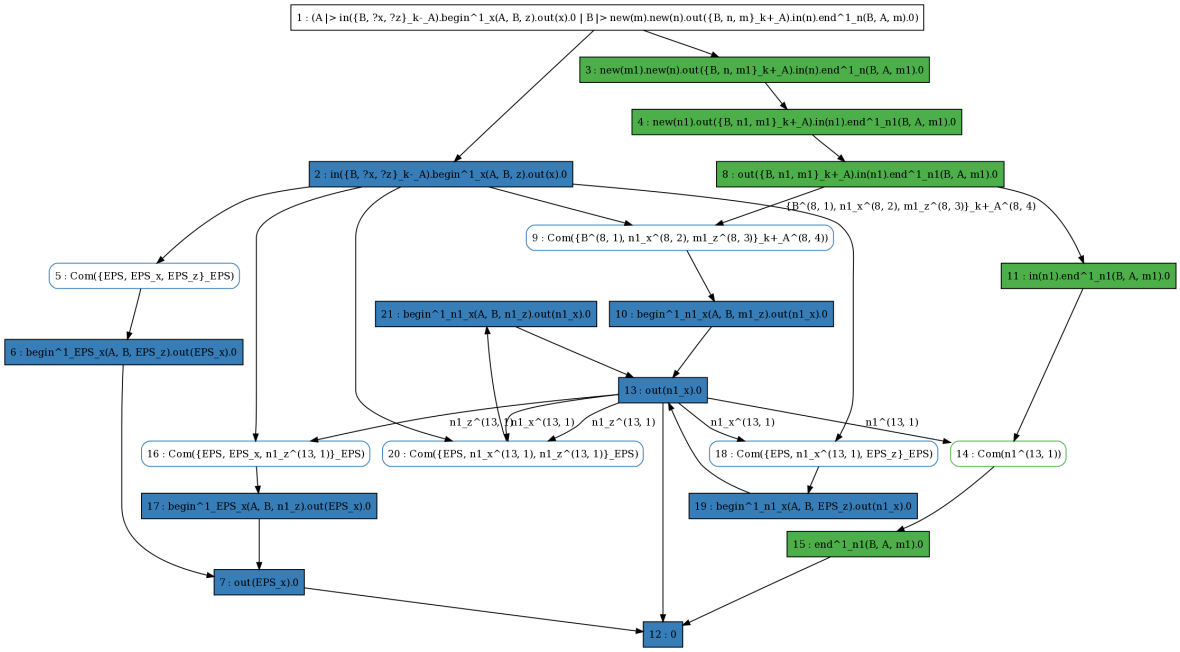


Figure A.1: Graph for protocol 2.1 with colored threads

$$\left\{\begin{array}{l} new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\ new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\ \quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\ in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \end{array}\right\}$$

$$\left\{\begin{array}{l} new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\ new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\ \quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\ \quad :: in_{com}\left(n1^{13,1}, n1\right) \\ new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\ \quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\ \quad :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\ in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\ \quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\ in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\ \quad :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\ \quad :: in_{com}\left(n1^{13,1}, n1\right) \\ in\left(\{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\ in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\ \quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \end{array}\right\}$$

Figure A.2: Traces of the `end`-assertion from example 2.1 - part 1

$$
\left\{
\begin{array}{l}
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^1\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^1\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^1\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^1\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right)
\end{array}
\right\}
$$

$$
\left\{
\begin{array}{l}
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^1\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^1\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^1\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right)
\end{array}
\right\}
$$

Figure A.3: Traces of the `end`-assertion from example 2.1 - part 2

$$
\left\{
\begin{array}{l}
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right)
\end{array}
\right\}
$$

Figure A.4: Traces of the `end`-assertion from example 2.1 - part 3

$$
\left\{
\begin{array}{l}
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right)
\end{array}
\right\}
$$

Figure A.5: Traces of the `end`-assertion from example 2.1 - part 4

$$
\left\{
\begin{array}{l}
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in\,(n1) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
new\,(m1) :: new\,(n1) :: out\left(\{B, n1, m1\}_{k_A^+}\right) :: in_{com}\left(\left\{B^{8,1}, n1_{(x)}^{8,2}, m1_{(z)}^{8,3}\right\}_{k_A^{+8,4}}, \{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) \\
\quad :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, m1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, m1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(z)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right) \\
in\left(\{B, n1_{(x)}, \mathscr{E}_{(z)}\}_{k_A^-}\right) :: begin_{n1_{(x)}}^{1}\left(A, B, \mathscr{E}_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1_{(x)}^{13,1}, \{B, n1_{(x)}, n1_{(z)}\}_{k_A^-}\right) \\
\quad :: begin_{n1_{(x)}}^{1}\left(A, B, n1_{(z)}\right) :: out\left(n1_{(x)}\right) :: in_{com}\left(n1^{13,1}, n1\right)
\end{array}
\right\}
$$

Figure A.6: Traces of the `end`-assertion from example 2.1 - part 5