Formal Specification of Security-relevant Properties of User Interfaces[1] (Extended Version)

Bernhard Beckert Gerd Beuster

{beckert|gb}@uni-koblenz.de

University Koblenz-Landau
Institut for Informatics

Abstract. When sensitive information is exchanged with the user of a computer system, the security of the system's user interface must be considered.
In this paper, we show how security relevant properties of a user interface can be modelled and specified using the Object Constraint Language (OCL). First, we demonstrate how the input-output functionality of an operating system can be modelled and formally specified. And second, using a text-based email system as an example, we explain how input-output-related security properties of an application can be modelled and formally specified (and later on verified).

1 Introduction

A large part of the specification of interactive applications is concerned with the relation between user input and the information shown to the user. For example, when editing a text, the current (internal) state of the text should be shown to the user, and user input should cause changes to the text. Usually, the specification of user input and system output is rather informal. Specifications declare that something "is shown on the screen" and the user "enters a text." In most cases, this informal description is sufficient. However, in security-critical applications, a precise and formal definition is desirable. In this paper, we show how security relevant properties of a user interface can be modelled, investigated, and ensured using formal methods.

First, in Chapter 3 we demonstrate how operating system requirements can be formalized for *guaranteeing* security against software-based man-in-the-middle attacks. Then, in Chapter 4, we show how security-relevant usability aspects of applications can be specified and—by proving that this specification is satisfied by the implementation—verified. In the Verisoft project (`www.verisoft.de`), our method is used to specify (and later on verify) an email system.

Though we only consider text-based user interfaces in this paper, our methods can be extended for handling graphical user interfaces.

[1] This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. See www.verisoft.de for more information about Verisoft.

1.1 Related Work

So far, works on secure interface design usually do not use formal methods, while formal methods for user interface specification do not take security aspects into consideration. Our approach extends previous work by bringing together formal methods for user interface specification with security-conscious design.

Abowd et al. [1] and Jain [7] give a survey of formal languages for the description of user interfaces. A well known model for this kind of interactive behavior is the PIE model developed by Dix and Runciman [5]. In this model, system behavior is defined as a function from commands issued by the user to effects produced by the system. In case of a text-based user interface, the input is a sequence of keystrokes and the output are characters displayed on the screen.

Non-static aspects of a system are usually modeled by state charts. In case of text-based user interfaces, state transitions are triggered by user commands. Depending on the methods chosen to model the user interface, the effects of commands are either represented as states, or emitted as events [4].

In a number of works, formal specification methods like Z have been applied to user interface design. One of the first formal specifications of interactive components was the specification of a text editor in Z in Sufrin's paper "Formal specification of a display editor" [9]. Based on Sufrin's specification, Booth and Jones implemented an editor in the Miranda functional programming language [2].

Besides formalizing user interface specification, two other aspects are important to our work. Interfaces for Human Computer Interaction (HCI) should follow good design practices, as summarized e.g. in ISO 9241 [6], and security considerations as given in [13]. For our work, we consider it most important that (1) at all times the user is aware of the state of the system in all aspects relevant to security, and that (2) the user gets clear feedback about the results of his or her activities.

One of the few works on secure interface design for application programs is Whitten and Tygar's case study about the user interface of PGP 5.0 [12]. However, the focus of Whitten and Tygar is different to ours. They examine whether the actual interface of a concrete application is suitable under security aspects, while we are interested in how to formally specify the required properties.

2 Environment and Notation

We use the Object Constraint Language (OCL) for specification. The OCL constraints given in this paper should be understandable without deeper knowledge of OCL.[2] When object attributes and variables are referred to in an OCL postcondition, the postfix "@pre" refers to the value of the attribute/variable in the initial state before the function was called. See [10, 11] for more information on OCL and [8] for the current language specification.

[2] To make the constraints easier to understand for readers not familiar with OCL, we sometimes use the standard mathematical notation instead of the OCL notation. For example, we use $x \in list$ instead of $list \rightarrow contains(x)$.

In this paper, we model a text-based user interface. Input comes from the keyboard and output goes to a terminal with a fixed number of rows and columns for display of characters. Assuming no additional input from other sources (like a mouse or network card), the behavior of a text-based application can be described as a function from a (finite) sequence of keystrokes to a screen output. That is, the behavior is specified by what is supposed to appear on the screen after a particular sequence of keystrokes. In a text-based application with a fixed screen size, screen output is a two dimensional array of characters. We use *keyboard* to refer to keyboard input and *screenAt* to refer to screen output. When we want to refer to a specific screen position, we use the notation $screenAt[x, y]$ for the character shown at screen position (x, y). Accordingly, *cursorAt* refers to the position of the cursor.

To refer to keyboard input up to resp. screen output at a particular point t in time, we use $keyboard(t)$ to denote the list of keystrokes entered up to time t, $screenAt(t)$ to denote the screen output at time t, and $cursorAt(t)$ to denote the position of the cursor at time t.

In a post-condition, t refers to the current time, i.e., the point in time when the function terminates, while $t@pre$ refers to the point in time when the function is entered. In this, we follow the common OCL syntax (though standard OCL does usually not contain explicit references to particular points in time).

When we specify program functions, the return value of the function can be an error code, indicating whether the operation was successful. As usual in OCL, we refer to the result of a function call by "result" in post-conditions.

OCL has the shortcoming that it does not make any assumptions about system properties that are not explicitly modeled (frame problem). To solve this problem, we (implicitly) add the following to our specifications: All functions cause only those effects explicitly mentioned.

When functions refer to one- or two-dimensional lists or strings, we use the usual []-notation to refer to elements. That is, $string[0]$ is the first character of *string*, $string[1]$ is the second, and so on.

A number of auxiliary functions are used to refer to certain properties of the system. These auxiliary functions are defined in Table 1.

3 Specifying Operating System Requirements

3.1 Overview

The operating system provides interfaces between application programs and the hardware. In the case of simple, text-based user interfaces, which we are examining in this paper, the operating system has to provide access to two resources: the keyboard and the screen. Most work on secure interface design assumes that the application runs in a safe and friendly environment. Although some work takes attacks on input/output facilities from the outside into account, interference with the input/output facilities from within the system (by trojans, worms, viruses, etc.) is usually not part of the attack scenarios. Here, we assume that the security critical application is running in a multi-process environment, where hostile processes may launch attacks on input/output facilities. Therefore,

Name	Description	First used in
$screenWidth()$	Width of screen	Chapter 3.2
$screenHeight()$	Height of screen	Chapter 3.2
$stringAt(t)[x,y] = s$	Boolean function returning true if for all $0 \leq n < \|s\|$: $screenAt(t)[x+n,y] = s[n]$	Chapter 3.2
$key(c)$	The character code of c, where c is an element of the list *keyboard* of keystrokes	Chapter 3.3
$timestamp(c)$	The time at which an element c of the list *keyboard* of keystrokes was entered	Chapter 3.3

Table 1. Auxiliary functions

we provide software-based criteria for trusted input/output facilities. The "security perimeter", i.e., the boundary of the secure part of the system is pushed outwards. We provide a formal method that *guarantees* security against software based man-in-the-middle attacks. Note, that our approach does not help against hardware-based attacks like faked keyboards placed by the attacker on top of the actual keyboard.

At this point we are not concerned with *secrecy*. We do not guarantee that no information is leaked to a third party. We only guarantee *observability*, i.e., all security-relevant aspects of the system are visible to the user.

3.2 Specifying Screen Output Functions

Below, we give constraints specifying the operating system functions for accessing the screen. These constraints use the functions *screenAt*, *cursorAt* (as described in the previous section), and the auxiliary functions from Table 1.

context `setChar(character, x,y)`
post **if** ((`x` ≥ 0) and (`x` $<$ $screenWidth()$) and
(`y` ≥ 0) and (`y` $<$ $screenHeight()$))
then
$screenAt(t)$[`x`, `y`] = `character` and
result = CHAR_SET_OK
else
result = POSITION_OUT_OF_BOUNDS
endif

```
context setCursor(x,y)
post   if  ((x ≥ 0) and (x < screenWidth()) and
            (y ≥ 0) and (y < screenHeight()))
       then
         cursorAt(t) = (x, y)  and
         result = CURSOR_SET_OK
       else
         result = POSITION_OUT_OF_BOUNDS
       endif
```

3.3 Specifying Keyboard Input Functions

Since user input comes from the keyboard, we can identify all user input during the lifetime of the application with a list of keystrokes, where a keystroke is a character (a character code) associated with a timestamp. As described in Chapter 2, $keyboard(t)$ denotes the list of keystrokes entered up to time t.

Usually, computer systems have an input buffer. This buffer is filled with the user's keystrokes independently of the current application's activity. When the application calls the operating system function for retrieving the next keystroke, the first keystroke of the keyboard buffer is returned. In the scenarios we are modeling, however, the use of a keyboard buffer is often not advisable. From the view of security, we want that the user approves or denies an activity only *after* he or she is aware of the options available. With a keyboard buffer, a user may enter commands that are executed at a later point in time. It could then happen that the user approves or denies an activity *before* the available options are shown to him or her. Therefore, we define the operating system function `getkeystroke` without using an input buffer. The function `getkeystroke` is specified to return the next character typed *after* its invocation.

Note, however, that it is possible to add operating system functions that make use of a keyboard buffer, or to make the keyboard buffer an explicit part of the application. For this, one would define two queues, one containing the keystrokes processed up to time t, and one containing the pending keystrokes at time t.

We use the auxiliary functions *key* and *timestamp* from Table 1 to access the character (code) of the keystrokes resp. the time when it was received.

```
context getkeystroke()
post   result ∈ keyboard(t)  and
       timestamp(result) > t@pre  and
       not ∃k ∈ keyboard(t) :
            (timestamp(k) > t@pre and
             timestamp(k) < timestamp(result))
```

3.4 Specifying Security-relevant Properties

Under security aspects, a key requirement for a system using keyboard input and screen output is the impossibility of man-in-the-middle attacks against the

keyboard and the screen. If an attacker can get in between the legitimate application and its input/output facilities, the attacker can manipulate the user at will.

There is no easy way to prevent physical man-in-the-middle attacks like, for example, covering the real keyboard with a faked keyboard as described in [3]. However, the prevention of software-based attacks with trojans, worms, viruses etc. is possible if the operating system provides means to guarantee exclusive access to the keyboard and screen. We call the process of acquiring exclusive access "locking" and the release of the lock "unlocking."

We consider information on whether screen and keyboard are locked and by which process to be part of the current configuration (i.e., the status) of the operating system. In the specification of requirements for the operating system, one has to refer to this information and other configuration details. For that purpose, we assume the relevant parts of the operating system configuration to be stored in a data structure (a class) `OSConf` with the following class attributes:

```
OSConf.screenLocked
OSConf.keyboardLocked
OSConf.ioStatus
```

`OSConf.screenLocked` and `OSConf.keyboardLocked` contain the process IDs (PIDs) of the processes locking the screen resp. the keyboard. A PID of 0 means that the resource is not locked. The third attribute `OSConf.ioStatus` can have the values `busy` and `waiting`. It indicates whether the system is busy or is waiting for input. While it is busy, all input is discarded (see comment about input buffers in Chapter 3.3).

Locking a resource is not sufficient to guarantee security. The user must also *know* which process locks a resource and whether the system is busy or not. Therefore, the operating system configuration must be shown to the user represented by a string of characters. We assume this string representation to be given by the function

$$\mathit{OSConfString} \ : \ \texttt{OSConf} \rightarrow \texttt{String} \ ,$$

which we do not further specify here. It must return a string that allows the user to determine the exact operating system configuration. Its actual implementation depends, for example, on the language(s) the user is supposed to understand.

We assume that the first line of the screen is reserved for information on the operating system configuration, i.e., the first line should be identical to $\mathit{OSConfString}(\texttt{OSConf})$. By not allowing processes (other than the operating system) access to row 0, the changed specifications of `setChar` and `setCursor` given below assure that the configuration information cannot be overwritten by any user applications nor attacking processes.

We specify the correct display of the operating configuration resources as an invariant of `OSConf`:

context `OSConf`
inv $\mathit{stringAt}(t)[0,0] = \mathit{OSConfString}(\texttt{OSConf})$

In the following constraints, `PID` refers to the PID of the current application. We start by specifying operating system functions for locking and unlocking screen and keyboard:

```
context lockscreen()
post    if OSConf.screenLocked@pre = 0
        then
          OSConf.screenLocked = PID  and
          result = SCREEN_LOCKED_OK
        else
          result = SCREEN_LOCKED_BY_OTHER
        endif
```

```
context unlockscreen()
post    if OSConf.screenLocked@pre = PID
        then
          OSConf.screenLocked = 0  and
          result = SCREEN_UNLOCKED_OK
        else
          result = SCREEN_LOCKED_BY_OTHER
        endif
```

```
context lockkeyboard()
post    if OSConf.keyboardLocked@pre = 0
        then
          OSConf.keyboardLocked = PID  and
          result = KEYBOARD_LOCKED_OK
        else
          result = KEYBOARD_LOCKED_BY_OTHER
        endif
```

```
context unlockkeyboard()
post    if OSConf.keyboardLocked@pre = PID
        then
          OSConf.keyboardLocked = 0  and
          result = KEYBOARD_UNLOCKED_OK
        else
          result = KEYBOARD_LOCKED_BY_OTHER
        endif
```

Now we proceed to re-specify `setChar`, `setCursor`, and `getkeystroke` to obey the locking mechanism and to protect the status line (line 0 of the screen).

```
context setChar(character, x,y)
post    if not OSConf.screenLocked = PID
        then
          result = SCREEN_LOCKED_BY_OTHER
        else if ((x ≥ 0) and (x < screenWidth()) and
                 (y ≥ 1) and (y < screenHeight()))
        then
          screenAt(t)[x, y] = character  and
          result = CHAR_SET_OK
        else
          result = POSITION_OUT_OF_BOUNDS
        endif
```

```
context setCursor(x,y)
post    if not OSConf.screenLocked = PID
        then
          result = SCREEN_LOCKED_BY_OTHER
        else if ((x ≥ 0) and (x < screenWidth()) and
                 (y ≥ 1) and (y < screenHeight()))
        then
          cursorAt(t) = (x, y)  and
          result = CURSOR_SET_OK
        else
          result = POSITION_OUT_OF_BOUNDS
        endif
```

```
context getkeystroke()
post    if not OSConf.keyboardLocked = PID
        then
          result = KEYBOARD_LOCKED_BY_OTHER
        else
          result ∈ keyboard(t)  and
          timestamp(result) > t@pre  and
          not ∃k ∈ keyboard(t) :
                (timestamp(k) > t@pre and
                 timestamp(k) < timestamp(result))
        endif
```

4 Security of Interactive Applications

4.1 Overview

In Chapter 3, we showed how to specify security-relevant properties of input/output functions provided by an operating system. By ensuring and verifying these properties, certain types of software-based man-in-the-middle attacks can be prevented.

There are, however, other essential aspects of a secure software system. In this chapter, we are going to introduce a method for specifying properties of applications that are desirable both for security and usability. Namely, the following properties are considered:

1. The user is always aware of the state of the system.
2. User input is only possible if the screen output is consistent.
3. Results of user actions are communicated to the user.

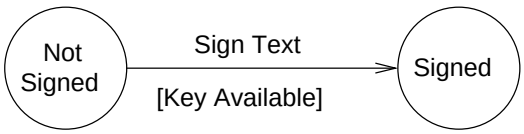

Fig. 1. State Chart Example

On an abstract level, the behavior of text-based interactive applications can be described using state charts. Edges are labeled with keystrokes, guard conditions, or both, as shown in Figure 1. In this example, the system transits from state Not Sent to state Sent if the command "Send Email" is issued and the guard condition "Message not Empty" is satisfied. Of course, the states in such as state chart are abstractions of the application's actual internal configuration, which is much richer in detail. Nevertheless, we assume that these states are the *right* abstraction in that the user has sufficient information about the internal configuration of the application if he or she knows in what abstract state the application is. Since, as said above, we also want the user to know what the result of the last issued command was, we define the configuration `Conf` of the application to contain—besides an application-dependend part `Conf.applicConf`—the last issued command `Conf.command`, and the result `Conf.commandResult` of that command, which can take the special valued `none` if the command is not yet completed (see Table 2).

`Conf.command`	Last issued command
`Conf.commandResult`	Result of last command
`Conf.applicConf`	Application-specific part of configuration

Table 2. Configuration of an application.

Now, two aspects of the application have to be specified:

1. The way in which the configuration is related to screen output; and how keyboard input corresponds to commands.
2. The effect that the execution of a command has, which must implement the abstract behavior specified by the state chart.

4.2 Specification of Input/Output Behavior

For the specifiation of the first aspect (input and output), we assume the following to be given (see Table 3):

- *stateAsString*(`state`) is a string that allows the user to determine what the state of the application is.
- *resultAsString*(`commandResult`) is a string that allows the user to determine what the result of the last issued command is.
- *screenOutput*(`applicConf`) is a two-dimensional array of characters. It contains the correct screen output corresponding to `applicConf`. Its dimensions are $screenWidth()$ and $screenHeight() - 3$.
- *command*(`char`) is the command that is issued by entering `char` on the keyboard.

We demand that *stateAsString*(`state`) is shown on the second line of the screen, and *resultAsString*(`commandResult`) on the third line (remember that the first line is reserved for the operating system's status line), which is why *screenOutput*(`applicConf`) must have a height of $screenHeight() - 3$.

Name	**Description**
stateAsString	Textual representation of the state
resultAsString	Textual representation of a command result
screenOutput	Screen output for a configuration
command	Command issued by entering a character
state	State abstraction of a configuration
newState	Next state when a command is issued in a certain configuration
result	Result of a command in a certain configuration

Table 3. Functions specifying an application.

Thus, the function `updateScreen` can be specified as follows. It is the application's function for updating the screen contents (using the operating system function `setChar`).

context `updateScreen()`
post $stringAt(t)[0,1] =$
$\quad stateAsString($`Conf.state`$)$ and
$stringAt(t)[0,2] =$
$\quad resultAsString($`Conf.commandResult`$)$ and
$\forall k \in \{3, \ldots, screenHeight() - 1\}$:
$\quad stringAt(t)[0,k] =$
$\quad\quad screenOutput($`Conf.applicConf`$)$[k]

4.3 Specification of Command Execution

For the specification of the second aspect (command execution), we assume the following to be given (see Table 3):

– *state*(`applicConf`) is the state abstraction of the application configuration.
– *newState*(`applicConf`, `command`) specifies the state transition. (It has `applicConf` as an argument and not, as one might expect, the abstraction *state*(`applicConf`), because it depends on guard conditions that can only be evaluated using the concrete application configuration.
– *result*(`applicConf`, `command`) is the result of executing `command` when the application is in configuration `applicConf`.

Now, the function `execute` can be specified. It executes a command and implements the state transition by changing the application configuration.

```
context execute()
post   state(Conf.applicConf) =
           newState(Conf.applicConf@pre,
                    Conf.command@pre)
       Conf.commandResult =
           result(Conf.applicConf@pre,
                  Conf.command@pre)
```

4.4 The Application's Main Algorithm

Now, we have everything at hand to describe how the main algorithm of the application works: First, screen and keyboard are locked. Then, in the main loop, commands are read and executed while keeping the screen updated. These steps are arranged in the following way:

– Screen and keyboard are locked immediately on program start and unlocked when the program quits. If locking the screen or the keyboard fails, the program terminates.
– Whenever the program is waiting for user input, the screen is up to date. Commands can be issued only when the system is waiting. All keystrokes entered during processing are discarded. By this we ensure that the user issues a command only when the current configuration of the system is visible on the screen.
– When processing is finished, the loop starts over again unless the user has issued the command "quit."

A state chart for the main execution loop is shown in Figure 2. Corresponding pseudo is given in Algorithm 1.

The consistency of the screen output follows from the algorithm and the specification of `getkeystroke`. The screen is up to date when the system is waiting for user input, and immediately after user input, and it may be inconsistent in between. Since the operating system displays status information "waiting" when the system is waiting for user input and "busy" when it is not, the user knows when the display must be consistent (whenever the system is waiting for user input). The situation would become more complicated if we used an input buffer. In that case, there is no longer a direct relationship between waiting/busy status and the consistency of screen output. It would be necessary to show an extra "consistency flag" on the screen.

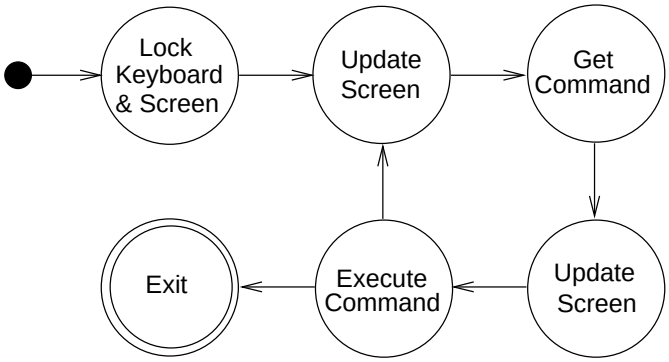

Fig. 2. State chart for the application's main algorithm (see Algorithm 1).

Algorithm 1 The application's main algorithm

```
 1: if not (lockkeyboard() = KEYBOARD_LOCKED_OK) then
 2:    Exit
 3: end if
 4: if not (lockscreen() = SCREEN_LOCKED_OK) then
 5:    Exit
 6: end if
 7: {OSConf.screenLocked = PID and OSConf.keyboardLocked = PID}
 8: repeat
 9:    updateScreen()
10:    Conf.command = command(key(getkeystroke()))
11:    Conf.commandResult = none
12:    updateScreen()
13:    execute()
14: until Conf.command = QUIT
```

5 Conclusions and Future Work

In Chapter 3 we gave a formal specification for text-based input/output functions of an operating system. This formalism can be extended to other input/output devices, e.g., card readers and graphical terminals. Additionally, we showed how to protect against software-based attacks on input/output resources. These security measurements require special functionality of the operating system. It must be able to grant processes exclusive access to input/output resources. Moreover, dedicated screen areas must be provided for information on who is locking the resources. This area must not be writable for anybody except the operating system.

The method we propose does not make any claims about what happens outside the realm of software. It cannot guarantee that an output device operates as intended, nor can it prevent tempering with the hardware of input/output devices.

In Chapter 4, we described a state-chart-based method for the formal specification of interactive applications. This formalism takes both security and usability aspects into consideration.

Our future work will go into two directions: As part of the Verisoft project (`www.verisoft.de`), the methods introduced in this paper are used to formally specify an email client. In Verisoft, both the operating system and the application program will be formally verified based on that specification.

The other direction of further work is to develop formal methods for the specification of applications that have richer user interfaces than a purely text based interface.

References

1. G. D. Abowd, J. P. Bowen, A. J. Dix, M. D. Harrison, and R. Took. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, October 1989.
2. S. P. Booth and S. B. Jones. A screen editor written in the miranda functional programming language. Technical Report TR-116, Department of Computing Science and Mathematics, University of Stirling, February 1994.
3. L. Bussard and Y. Roudier. Authentication in ubiquitous computing. In *UBICOMP 2002, Workshop on Security in Ubiquitous Computing*, Göteborg, Sweden, September 2002.
4. A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996.
5. A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *HCI'85: People and Computers I: Designing the Interface*, pages 13–22. Cambridge: Cambridge University Press, 1985.
6. International Organisation for Standardization. ISO 9241. Ergonomic requirements for office work with visual display terminals (VDTs), 1992–2001.
7. V. Jain. User interface description formalisms. Technical report, McGill University School of Computer Science, Montréal, Canada, 1994.
8. Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, Mar. 2003.

9. B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, pages 157–202, 1982.
10. J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, Mar. 1999.
11. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley Professional, 1998.
12. A. Whitten and J. Tygar. Usability of security: A case study. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1998.
13. K.-P. Yee. User interaction design for secure systems. In *Proceedings of the International Conference on Information and Communications Security*, Singapore, 2002. `www.sims.berkeley.edu/~ping/sid/uidss.pdf`.