# Reconstruction of Falsified Computer Logs for Digital Forensics Investigations

**Maolin Tang** **Colin Fidge**

Faculty of Science and Technology
Queensland University of Technology
Brisbane, Australia
Email: {m.tang, c.fidge}@qut.edu.au

**Abstract**

Digital forensics investigations aim to find evidence that helps confirm or disprove a hypothesis about an alleged computer-based crime. However, the ease with which computer-literate criminals can falsify computer event logs makes the prosecutor's job highly challenging. Given a log which is suspected to have been falsified or tampered with, a prosecutor is obliged to provide a convincing explanation for how the log may have been created. Here we focus on showing how a suspect computer event log can be transformed into a hypothesised actual sequence of events, consistent with independent, trusted sources of event orderings. We present two algorithms which allow the effort involved in falsifying logs to be quantified, as a function of the number of 'moves' required to transform the suspect log into the hypothesised one, thus allowing a prosecutor to assess the likelihood of a particular falsification scenario. The first algorithm always produces an optimal solution but, for reasons of efficiency, is suitable for short event logs only. To deal with the massive amount of data typically found in computer event logs, we also present a second heuristic algorithm which is considerably more efficient but may not always generate an optimal outcome.

*Keywords:* Digital forensics, computer logs, event correlation algorithms.

## 1 Introduction

Digital forensics involves investigations into suspected crimes or misbehaviors that are manifested in computer-based evidence (Richard III & Roussev 2006). Part of this process is showing that sequences of events hypothesised by a forensic investigator or legal prosecutor are consistent with the available digital evidence (Mohay 2005). If the integrity of the evidence, typically computer-generated logs, is in doubt, then the assumed actual sequence of events needs to be reconstructed, and its relationship to the digital artifacts needs to be explained in terms of actions that could reasonably have been performed by the defendant.

We wish to thank the anonymous reviewers for their many helpful comments. This work was supported in part by the Defence Signals Directorate and the Australian Research Council via ARC-Linkage Projects grant LP0776344.



Digital forensics has three major phases: *acquisition*, *analysis* and *presentation* (Carrier 2002). In the acquisition phase the state of a digital system is preserved so that it can be analysed later. Tools used in the acquisition phase are programs that can copy data and software from a suspect storage device to a trusted one. In the analysis phase, the acquired data is examined in order to find pieces of evidence to support or contradict a given hypothesis intended to explain how the evidence was created. In the presentation phase the conclusions from the analysis phase are presented in a legal setting in a way comprehensible to non-experts. Here we are concerned with the analysis phase, especially the process of explaining how deliberately falsified computer log evidence may have been created.

Analyses of computer logs usually rely on timestamps to determine the order in which events occurred. However, the clocks used to generate timestamps may be inaccurate when compared to 'absolute' time (Schatz et al. 2006). Even worse, computer-literate criminals may attempt to cover their tracks by adjusting the clocks on their computers to create misleading event logs (Willassen 2008*a*).

To overcome this, we need to know whether or not an acquired event log may have been falsified and, if so, how this could have been done. To some extent this problem can be approached by comparing the suspect event log with independently-generated, trusted event logs. The difficulty, however, is that event logs produced by different sources, e.g., computer operating systems, door swipecard readers, network firewalls, network routers, Internet Service Providers, web servers, etc, contain records of different kinds of events. Determining the relationship between these events, and even finding a common representation for them, can be highly challenging (Chen et al. 2003).

Previous work has focussed on determining which trusted, causally-related event orderings are consistent with the timestamped evidence, according to a particular falsification hypothesis (Willassen 2008*b*). However, the problem of reconstructing hypothesised event sequences from the available digital evidence has received relatively little attention.

In this paper we consider the problem of determining how a suspect event log may have been created, in the context of a sequence of events hypothesised from independent, trusted sources of information. This outcome allows a prosecutor to assess the likelihood of different assumed actions by the defendant. A proposed scenario which involves relatively little effort on the part of a suspected criminal is more likely than one which requires a large number of actions in order to produce the presumably-falsified computer log.

We present two algorithms for quantifying the number of steps required to turn a prosecutor's hypothesised sequence of actual events into the sequence

of events found in a seized computer's log. The first algorithm always produces an optimal result, in the sense that it completes the transformation in the fewest steps. However, because computer forensics typically involves analysing massive amounts of data (Mohay 2005), we also present a heuristic algorithm which is considerably more efficient but may not yield an optimal answer. Nevertheless, for the purposes of arguing a legal case, a sub-optimal solution will often suffice.

## 2 Previous and Related Work

Much work has already been done on the problem of how to analyse and 'correlate' events found in different data logs, but in general this work differs from ours because it does not consider the possibility of reordering the events in a suspect log to match a particular hypothesis.

Broadly speaking, the legal importance of computer logs is illustrated by the emergence of standards for their maintenance. For instance, Kent & Souppaya (2006) present general guidelines for responsible management of computer security logs, including storage, data formats, data integrity, data confidentiality, etc.

Numerous tools have been proposed to help forensic investigators process the large amounts of data produced by computer-based systems. Case et al. (2008) note that existing forensics tools provide isolated data mining functions but leave the job of interpreting and correlating the data to the human investigator. They present a prototype environment for integrating and correlating timestamped events obtained from several different data logs into a single sequence. To do this they rely on the timestamps in login files and network traces to reconstruct the actions performed by a user during a particular session. They do not, however, consider the possibility that the ordering implied by the timestamps is not the actual one.

Similarly, Best et al. (2004) devised a tool for analysis of operating system logs to help security auditors mine the generated data. The tool searches for attempts to circumvent security mechanisms and changes in users' behaviour, as compared to previously-accumulated user profiles. More recently, Raghavan et al. (2009) proposed a software architecture for integrating forensic data from multiple sources that allows an investigator to explore theories about past behaviours, but again it does not consider the possibility of event reordering.

Since forensic evidence often relates to causal relationships between events, much previous work has focussed on detecting causal relationships across different logs. For instance, in the context of distributed systems, Gazagnaire & Hélouët (2007) considered the problem of knowing whether or not two events in different logs are causally related. Since logged data is typically incomplete with respect to event causality, they developed a theory for composition of partially-ordered event sets that do not include all causal relationships between events. Their 'event correlation' theory allows missing causal relationships to be reconstructed. However, they assumed that the causal relationships present in the given logs are correct, unlike our situation.

In a related vein, Schatz et al. (2004) focussed on the semantic content of event logs, so that domain-specific inferences can be drawn about possible hypotheses using data other than that which is obviously security-related. They used pattern-based reasoning rules to correlate logged events so that causal relationships between events can be identified. They subsequently extended this work so that the reasoning rules could infer information from multiple heterogenous domains, e.g., firewall logs and swipecard access logs (Schatz et al. 2005). The motivation for this work is similar to our own, but again they assume that the ordering of events in the individual logs being correlated is accurate, and make no allowance for deliberate falsification of event orderings.

In practice, the causal ordering of logged events is assumed to be determined by their associated timestamps, so many studies have focussed on the accuracy of timestamping mechanisms. For instance, Boyd & Forster (2004) noted that analysis of timestamped events can be complex due to different date-time data formats and time zones. They cite a criminal case in which the defence asserted that police had falsified evidence by adding files to the defendant's computer after it had been seized. Ultimately it transpired that this was not the case—the defence's expert witness had, in fact, misinterpreted the ordering of events by failing to include a necessary timezone-related offset to the timestamps.

Gladyshev & Patel (2005) presented a formalisation of the problem of placing bounds on the range of times within which a non-timestamped event could have occurred, by 'sandwiching' it between two causally-related timestamped events.

Willassen (2008*b*) treated the problem of inaccurate timestamps as the need to test a 'clock hypothesis' against the timestamped evidence. The aim of this work was to determine which causal action sequences are possible, given the hypothesised behaviour of the clock used to generate the timestamps. This work was then extended so that it could be used to detect 'antedating', i.e., deliberate falsification of timestamps, by detecting that the timestamps on events are inconsistent with (known) necessary causal orderings (Willassen 2008*a*). This work is highly relevant to our own, but its aim was to detect mismatches between actual and timestamped event sequences, whereas we are concerned with how to transform one to the other.

Similarly, Schatz et al. (2006) considered the problem of correlating timestamped events when the clocks used to generate the timestamps differ from absolute time due to inadvertent clock skew or drift, or deliberate clock tampering. They present the results of empirical experiments for measuring the offset of the timestamping clock from a trusted reference clock in order to determine the likely offset in the timestamps. Again, however, the way in which actual and timestamped event orderings can be linked was not the primary focus of their work.

Finally, to a certain extent, the algorithms we develop below are related to the classic 'marriage problem', in which the most efficient pairing of elements from two disjoint sets must be found. This well-known combinatorial challenge is usually solved via a backtracking algorithm (Berman & Paul 2005). It differs from our problem, however, because there is no concept of an overall ordering in the two sets of elements.

## 3 Problem Statement

Consider the problem faced by a prosecutor intent on showing that acquired evidence is consistent with a particular hypothesis concerning criminal behaviour, even in the face of deliberate falsification of the evidence. In the case of digital data this usually means attempting to reconcile a hypothesis formed from trusted information, such as the logs generated by an Internet Service Provider, with non-trustworthy information, such as the file and event timestamps on

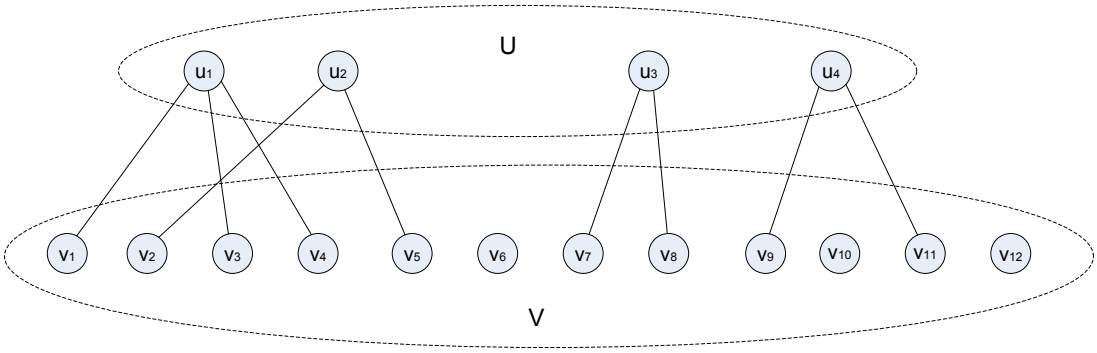


Figure 1: Example computer log and ISP log correlation model

the personal computer owned by the accused (Schatz et al. 2006).

For example, the correlation between the events in a computer log and the events in a corresponding ISP log can be modelled as a bigraph $G = (U \cup V, E)$, where $V$ is the *computer event set*, containing all the events in the potentially-compromised computer's logs, and $U$ is the *ISP event set*, containing all relevant events in the trustworthy ISP's event logs. An edge $(u, v)$ is in $E$ if event $v \in V$ is correlated to event $u \in U$, or if event $v \in U$ is correlated to event $u \in V$.

Each event in the ISP's event log may be related to multiple events in the computer's activity log. For example, an event in the ISP log may be a visit to a particular website from the defendant's personal computer. While visiting the website, however, multiple individual files may have been downloaded to the computer. Thus, in the computer log there will be multiple download events, while the ISP log records only a single website visit.

Figure 1 illustrates such a computer log and ISP log correlation model. In this case there are four events recorded in the ISP's log, $u_1$, $u_2$, $u_3$ and $u_4$, but there are twelve events in the personal computer's log, $v_1$, $v_2$, $\cdots$, $v_{12}$. Events $v_1$, $v_3$ and $v_4$ recorded in the computer's log are correlated to event $u_1$ in the ISP's log; events $v_2$ and $v_5$ in the computer's log are related to event $u_2$ in the ISP's log; events $v_7$ and $v_8$ in the computer's log are related to event $u_3$ in the ISP's log; and events $v_9$ and $v_{11}$ in the computer's log are related to event $u_4$ in the ISP's log. Computer log events $v_6$, $v_{10}$ and $v_{12}$ are generated by local actions that did not involve Internet access, so they have no correspondence to any ISP events.

We assume the events in the ISP's log were all timestamped, in a way accurately reflecting the order in which Internet-related events actually occurred. This ordering can be modeled by a directed graph $G_U = (V_U, E_U)$, where $V_U = U$ is the set of ISP events, and $(u_i, u_j) \in E_U$ if $u_i$ and $u_j$ were two consecutive events in the ISP's log. We call directed graph $G_U$ the *ISP event constraint graph*. For instance, assume the events in the ISP's log in Figure 1 have timestamps corresponding to the sequence $u_3$, $u_2$, $u_1$ and $u_4$. Then the ISP event constraint graph is shown in Figure 2.

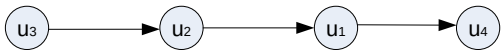


Figure 2: Example ISP event constraint graph

An event in the ISP's event log may be correlated to multiple events in the corresponding computer event log and there may be reasonableness constraints between the events. For example, a computer file must be created before it can be accessed or modified. Thus, a file creation event must occur before an access or modification event for the same file. Such constraints can be represented by a *computer event constraint graph* $G_V = (V_V, E_V)$, where $V_V = V$ is the computer event set, and $(v_i, v_j) \in E_V$ if event $v_i$ must occur before event $v_j$. Figure 3 is an instance of the computer event constraint graph for the set of computer events shown in Figure 1.

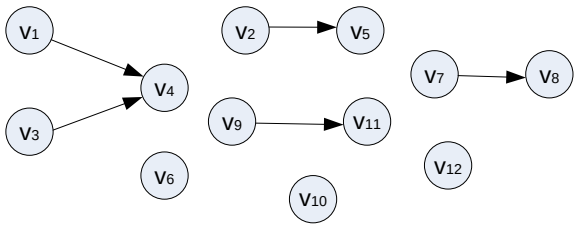


Figure 3: Example computer event constraint graph

All of the information in Figures 1, 2 and 3 can be used by a prosecutor to form a hypothesis about the actual sequence of events that must have occurred in some criminal case. If the log of events recorded on the defendant's computer is inconsistent with these independent sources of information, we are obliged to find an explanation for how the presumably-falsified log could have been created. This is the motivation for our research.

It is assumed that all the events in the computer event log were timestamped. We define a computer event log as a sequence of recorded events ordered by their timestamps. Such a log is considered to be 'falsified' if the timestamped event sequence violates any of the event orderings implied by independent information sources such as those in Figures 1, 2 and 3. For example, the timestamped event sequence $\langle v_1, v_9, v_2, v_4, v_3, v_5, v_6, v_7, v_8, v_{10}, v_{11}, v_{12} \rangle$ is falsified because it violates the constraint that event $v_3$ must occur before event $v_4$ in Figure 3, and the requirement that $v_5$ comes before $v_9$ as required by Figures 1 and 2.

When someone attempts to disguise their actions by falsifying a computer log, this is often done by resetting the clock on their computer to alter the apparent order in which certain actions were actually performed, and by then setting the clock back again to hide the fact that any such subterfuge has been

attempted (Willassen 2008*a*). This has the effect of causing a *consecutive sequence* of events to be given misleading timestamps. If the timestamps on events are used to infer the assumed event ordering, the outcome is that the whole sequence of events performed while the computer's clock was maladjusted will appear in the wrong place in the overall event history. In effect, this mechanism can be used by malefactors to "move" sequences of events in the log.

Given the suspicion that such a deception has taken place, we assume that a forensic investigator (or criminal prosecutor) has a hypothesis in mind about the actual sequence of events, typically informed by independent sources of information such as ISP logs. Our technical goal, therefore, given a purportedly-falsified computer event log, and the forensic investigator's hypothesised computer event log, is to assess the reasonableness of the forensic investigator's hypothesis by showing how the hypothesised log can be transformed into the falsified one in the smallest number of moves.

## 4 Our Approach

This section presents two algorithms for the falsified computer event log reconstruction problem, an optimal algorithm for analysing relatively small data sets, and a heuristic algorithm for large data sets. The latter is necessary in practice because the problem of reconstructing event logs is a combinatorial optimisation problem and the search space of the optimal algorithm has a super-linear order of growth.

### 4.1 Definitions

Before presenting the algorithms themselves, we introduce some definitions used in the rest of the paper.

A *consecutive event sequence* is a sequence of events in a falsified computer event log that occur consecutively in the corresponding hypothesised computer event log. A *maximal consecutive event sequence* is a consecutive event sequence that is not covered by any other consecutive event sequence. For example, let $S = v_4v_5v_1v_2v_3$ be a falsified computer event log and $S^* = v_1v_2v_3v_4v_5$ be the hypothesised computer event log. Then, $\langle v_1\rangle$, $\langle v_1v_2\rangle$ and $\langle v_1v_2v_3\rangle$ are three consecutive event sequences in $S$. But only the consecutive sequence $\langle v_1v_2v_3\rangle$ is a maximal consecutive computer event sequence. There are two maximal consecutive event sequences in $S$, $\langle v_1v_2v_3\rangle$ and $\langle v_4v_5\rangle$.

In order to facilitate the presentation, we assume that the computer events in a hypothesised computer event log are in sequential order according to their subscripts, without loss of generality. Thus, a maximal consecutive computer event sequence in $S$ must be of the form $\langle v_iv_{i+1}\cdots v_j\rangle$ and is denoted as $v_{i-j}$ in the rest of this paper, where $i \leq j$.

A falsified computer event log can be represented by a sequence of maximal consecutive event sequences. For example, $S = v_4v_5v_1v_2v_3$ can be represented as $v_{4-5}v_{1-3}$.

### 4.2 Finding Maximal Consecutive Event Sequences

To start our analysis we first partition the falsified event sequence $S$ into a sequence of maximal consecutive event sequences, so that these entire sequences can be 'moved' as a single unit. This is necessary since a single adjustment to a computer's clock will effectively shift an entire sequence of subsequent events in the log. However, we do not want to consider each of the events individually, as this would give a misleading impression about the 'effort' required to falsify the log.

**Algorithm 1** Transform a computer event sequence into a sequence of maximal consecutive event sequences

**Require:** A computer event sequence $S = v_1v_2\cdots v_n$, and its corresponding hypothesised computer event sequence $S^* = v_1^*v_2^*\cdots v_n^*$;
**Ensure:** The maximal consecutive event sequence representation of $S$.
$i = 1$;
$seqs = \emptyset$;
**while** $i \leq n$ **do**
  Find the maximal consecutive event sequence in $S$ starting from the $i^{\text{th}}$ computer event using Algorithm 2;
  $i = i +$ the length of the maximal consecutive event sequence;
  $seqs = seqs +$ the maximal consecutive event sequence;
**end while**;
**return** $seqs$.

Algorithm 1 transforms a computer event sequence into a sequence of maximal consecutive event sequences. Firstly, it finds the maximal consecutive computer event sequence in $S$ starting from $S$'s first event, then finds the next maximal consecutive computer event sequence starting from the end of the maximal sequence, and so on. The process of finding the next maximal consecutive computer event sequence is repeated until all maximal consecutive computer event sequences are found. The maximal consecutive computer event sequences are concatenated to form a sequence of maximal consecutive computer event sequences.

Algorithm 2 finds a maximal consecutive computer event sequence in $S$, starting from a particular event $v_i$, and is used as an auxiliary subroutine by Algorithm 1. It first searches through $S^*$ for the starting event $v_i$, and then accumulates events as long as those in $S$ and $S^*$ match.

**Algorithm 2** Find a maximal consecutive event sequence

**Require:** A computer event sequence $S = v_1v_2\cdots v_n$, its corresponding hypothesised computer event sequence $S^* = v_1^*v_2^*\cdots v_n^*$, and a starting location $i$;
**Ensure:** A maximal consecutive event sequence starting from location $i$.
$j = 1$;
**while** $v_i \neq v_j^*$ **do**
  $j = j + 1$;
**end while**;
$start\_location = j$;
**repeat**
  $i = i + 1$;
  $j = j + 1$;
**until** $v_i \neq v_j^*$;
$end\_location = j - 1$;
**return** $v_{start\_location-end\_location}$.

It is straightforward to show that the computational complexity of Algorithm 1 is $O(n \times m)$, where $n$ is the number of computer events in $S$ and $m$ is the number of maximal consecutive computer event sequences in $S$. Algorithm 2 takes $O(n)$ time to find the location of event $v_i$ in $S^*$ and $O(n)$ time to find the maximal computer event sequence starting from this event, so the computational complexity of Algo-

rithm 2 is $O(n)$. Assume that $S$ contains $m$ maximal consecutive computer event sequences. Then Algorithm 1 invokes Algorithm 2 $m$ times. Thus, the computational complexity of Algorithm 1 is $O(n \times m)$.

### 4.3 An Optimal Algorithm for Determining the Moves Required to Falsify a Log

This section presents an algorithm guaranteed to find the optimal solution to the problem of how a computer log could have been falsified, where optimality is defined as transforming the hypothesised log into the presumably-falsified one in the minimal number of moves. Each 'move' involves shifting an entire sequence of events and, in practice, is the consequence of resetting the computer's clock to disguise the true sequence of events.

The algorithm is basically an $A^*$ algorithm (Hart et al. 1968). An $A^*$ algorithm is a best-first graph searching algorithm that finds the shortest path from a given source node $s$ to a goal node $t$ in the graph. It uses a heuristic function, $f(x)$, to determine the order in which to visit nodes in the graph. The heuristic function is a sum of two subfunctions: a distance function, $g(x)$, which is the distance from the source node $s$ to current node $x$, and an admissible "heuristic estimate" $h(x)$ of the distance from node $x$ to goal node $t$. Function $h(x)$ must be an admissible heuristic, that is, it must not overestimate the distance from $x$ to $t$ in order to guarantee the admissibility and optimality of the $A^*$ algorithm (Hart et al. 1968).

The evaluation function used by our $A^*$ algorithm is defined by Equation 1 below.

$$f(x) = g(x) + h(x) \tag{1}$$

Here $g(x)$ is the actual distance (the number of moves) from $s$ to $x$, and $h(x)$ is defined by Equation 2.

$$h(x) = \lceil (m-1)/3 \rceil \tag{2}$$

Here $m$ stands for the number of maximal consecutive event sequences in $x$ and $\lceil m/3 \rceil$ is the smallest integer greater than or equal to $m/3$. Function $h(x)$ is an admissible heuristic estimate of the distance, i.e., the number of moves, from $x$ to $t$ because the number of moves from $x$ to $t$ is greater than or equal to $\lceil m/3 \rceil - 1$ (see Section 4.4).

Algorithm 3 is our $A^*$ search algorithm and Algorithm 4 is an auxiliary algorithm for backtracking and generating output.

In the $A^*$ algorithm, four data structures are used. The first is *OpenSet*, which is a set that stores computer event logs that are at the frontier of the $A^*$ search. The second is *ClosedSet*, which keeps computer event logs that have been visited. The third is $came\_from[x]$, which is used to store the parent computer event log sequence of $x$. For example, if $came\_from[x] = y$, then it indicates that $y$ was obtained by moving a maximal computer event sequence in $x$. The fourth is $moves[y]$, which is used to store how $y$ was obtained. Specifically, $moves[y]$ contains a maximal computer sequence and the location of the maximal computer sequence in its parent. For example, $moves[y] = [v_{i-j}, from, to]$ indicates that the parent computer event log of $y$ can be built by moving the maximal consecutive event sequence $v_{i-j}$ from location $from$ to location $to$ in $y$. Note that locations $from$ and $to$ are those in the original computer event log, rather than the locations in the maximal consecutive event sequence.

Consider a simple example to illustrate how the $A^*$ search algorithm works. Let the hypothesised event sequence be $S^* = v_1v_2v_3v_4v_5v_6v_7v_8$ and the falsified one be $S = v_3v_4v_5v_7v_8v_6v_1v_2$. First of all,

**Algorithm 3** $A^*$ search

**Require:** A falsified sequence of computer events $S$, and a hypothesised computer event log $S^*$;
**Ensure:** A sequence of moves for transforming $S^*$ into $S$.
Transform $S$ into a sequence of maximal consecutive computer event sequences using Algorithm 1;
$m$ = the number of maximal consecutive events in $S$;
**for** $i = 1, m$ **do**
  $came\_from[i] = null$;
  $moves[i] = null$;
**end for**;
$ClosedSet = \emptyset$;
$OpenSet = \{S\}$;
$g(S) = 0$;
$h(S) = \lceil (m-1)/3 \rceil$, where $m$ is the number of maximal consecutive event sequences in $S$;
$f(S) = g(S) + h(S)$;
**while** $OpenSet \neq \emptyset$ **do**
  $x$ = the element in $OpenSet$ that has the least $f(x) = g(x) + h(x)$ and contains the minimal number of maximal consecutive event sequences (if there is more than one element that satisfies the conditions, then the one that was added to $OpenSet$ first is selected);
  Remove $x$ from $OpenSet$;
  Add $x$ to $ClosedSet$;
  $m$ = the number of maximal consecutive computer event sequences in $x$;
  **if** $m = 1$ **then**
    Use Algorithm 4 to backtrack and output the sequence of moves from $S^*$ to $S$, and then stop.
  **end if**;
  **for** each $y$ that can be obtained by moving a maximal consecutive computer event sequence from location $i$ to location $j$, where $1 \leq i, j \leq m$ **do**
    **if** $y \notin ClosedSet$ **then**
      $new\_g(y) = g(x) + 1$;
      $came\_from[y] = x$;
      $moves[y] = [x, i, j]$;
      **if** $y \notin OpenSet$ **then**
        Add $y$ to $OpenSet$;
        $g(y) = new\_g(y)$;
        $h(y) = \lceil (m-1)/3 \rceil$, where $m$ is the number of maximal consecutive event sequences in $y$;
        $f(y) = g(y) + h(y)$;
      **else**
        **if** $new\_g(y) < g(y)$ **then**
          $g(y) = new\_g(y)$;
        **end if**
      **end if**
    **end if**
  **end for**
**end while**

**Algorithm 4** Backtracking moves

**Require:** A hypothesis computer event sequence $S^*$, a falsified computer event sequence $S$, and values $came\_from$ and $moves$ generated by the search algorithms;
**Ensure:** A sequence of moves that transform a hypothesised computer event sequence $S^*$ to a falsified computer event sequence $S$.
$x = S^*$;
$y = came\_from[x]$;
**while** $y \neq null$ **do**
  **print** $moves[x]$;
  $x = y$;
  $y = came\_from[x]$;
**end while**

the $A^*$ search algorithm transforms $S$ into a sequence of maximal consecutive computer event sequences $v_{3-5}v_{7-8}v_{6-6}v_{1-2}$. The number of maximal consecutive computer event sequences is $m = 4$.

After initializing variables $came\_from[i]$, $moves[i]$, $ClosedSet$ and $OpenSet$, the $A^*$ search algorithm explores all the neighbours of $S$. A neighbour of $S$ is defined as a computer event log that can be obtained by moving a maximal consecutive computer event sequence from one location to another in $S$. After initialisation, $OpenSet = \{S\}$ and $ClosedSet = \emptyset$, and $f(S)$ is calculated.

In the first exploration, the $A^*$ search algorithm selects $S$ to explore as it is the only element in $OpenSet$ and moves it from $OpenSet$ to $ClosedSet$. Then, all neighbours of $S$ are added to $OpenSet$ as none of them is in $OpenSet$ or $ClosedSet$, all the $f$ values of the neighbours are updated, and all the $came\_from$ and $moves$ elements are updated as well. The search space after this exploration is shown in Figure 4.

After the first exploration, $ClosedSet$ equals $\{S\}$ and $OpenSet$ is $\{v_{7-8}v_{3-6}v_{1-2}, v_{7-8}v_{6-6}v_{3-5}v_{1-2}, v_{7-8}v_{6-6}v_{1-5}, v_{3-8}v_{1-2}, v_{3-6}v_{1-2}v_{7-8}, v_{6-6}v_{3-5}v_{7-8}v_{1-2}, v_{3-5}v_{7-8}v_{1-2}v_{6-6}, v_{1-5}v_{7-8}v_{6-6}, v_{3-5}v_{1-2}v_{7-8}v_{6-6}\}$.

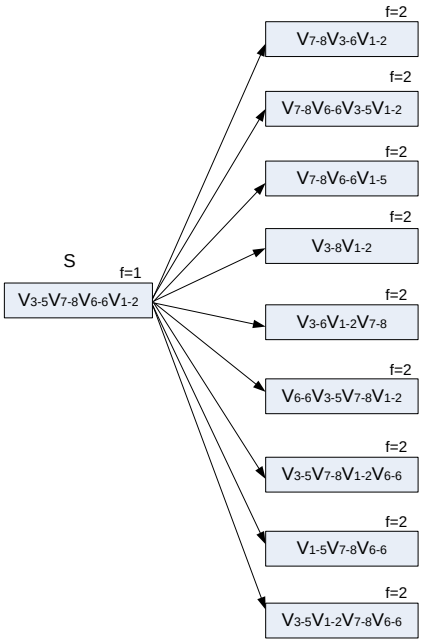


Figure 4: The $A^*$ search space after exploring $S$

In the second exploration, the $A^*$ search algorithm selects $v_{3-8}v_{1-2}$ to explore as it has the minimal $f$ value and contains a minimal number of maximal consecutive event sequences. The algorithm moves $v_{3-8}v_{1-2}$ from $OpenSet$ to $ClosedSet$. Then, $v_{1-8} = S^*$ is added to $OpenSet$ as $v_{1-8}$ is the only neighbour of $v_{3-8}v_{1-2}$ that is not in $OpenSet$ or $ClosedSet$, the $f$ value of $v_{1-8}$ is calculated, and the $came\_from$ and $moves$ elements are updated as well. The search space after this exploration is shown in Figure 5.

After the second exploration, $ClosedSet$ equals $\{S, v_{3-8}v_{1-2}\}$ and $OpenSet$ equals $\{v_{7-8}v_{3-6}v_{1-2}, v_{7-8}v_{6-6}v_{3-5}v_{1-2}, v_{7-8}v_{6-6}v_{1-5}, v_{3-6}v_{1-2}v_{7-8}, v_{6-6}v_{3-5}v_{7-8}v_{1-2}, v_{3-5}v_{7-8}v_{1-2}v_{6-6}, v_{1-5}v_{7-8}v_{6-6}, v_{3-5}v_{1-2}v_{7-8}v_{6-6}, v_{1-8}\}$.

In the third exploration, the $A^*$ search algorithm selects sequence $v_{1-8}$ to visit. Since $m = 1$, the $A^*$ search algorithm has found the goal. After the goal computer event log is found, the backtracking algorithm is used to retrieve the path from $S^*$ to $S$. This path is then the optimal sequence of moves, $[v_{3-8}, 3, 1]$ followed by $[v_{7-8}, 5, 4]$. In other words, event sequence $v_1v_2v_3v_4v_5v_6v_7v_8$ can be transformed into event sequence $v_3v_4v_5v_7v_8v_6v_1v_2$ in only two 'moves', highlighting the fact that this seemingly complex rearrangement of events does not take a major effort. In a criminal case this observation may be crucial in making the prosecutor's case convincing.

### 4.4 A Heuristic Algorithm for Determining the Moves Required to Falsify a Log

The $A^*$ algorithm can always find an optimal solution, i.e., a minimal sequence of moves that transform a hypothesised computer event log into a falsified one. Unfortunately, $A^*$ algorithms are notoriously expensive. Their time complexity ranges from polynomial to exponential, depending on the heuristic function used, and their space complexity is often exponential. Given the large number of events in actual computer logs, the algorithm in Section 4.3 will often prove impractical.

Therefore, this section presents a heuristic algorithm that is more efficient. Although it will always find a solution, it cannot be guaranteed to find the optimal one. Nevertheless, by making use of some contextual information, the heuristic algorithm will usually find a solution involving a small number of moves. Furthermore, in the context of a legal argument about the likelihood of a crime having been committed, finding a mathematically optimal solution may not be necessary.

Like the $A^*$ algorithm, it is assumed that a falsified computer event log was created by repeatedly moving one consecutive computer event sequence at a time from one location to another. In order to minimise the number of moves, we assume that whenever a consecutive computer event sequence is moved, it has to be a maximal consecutive computer event sequence. The algorithm's input again includes a falsified computer event log $S$ and a hypothesised computer event log $S^*$. The output of the algorithm is a sequence of moves of maximal consecutive computer event sequences that transforms $S^*$ into $S$.

The heuristic algorithm treats the log reconstruction problem as one of finding a shortest path in a state space. Each state in the state space represents a computer event sequence that can be transformed from $S$ by moving some maximal consecutive computer event sequences in $S$ in a particular order. Once the path is found, the heuristic algorithm backtracks along the shortest path, which represents the sequence of moves of maximal consecutive computer event sequences, using the same backtracking procedure used by the $A^*$ algorithm.

Algorithm 5 is our heuristic algorithm for the falsified computer event log reconstruction problem. The search process is iterative. In each iteration the algorithm explores all the new computer event sequences that can be obtained by moving one of the maximal consecutive event sequences in $S$ and chooses the new computer event sequence that contains the minimal number of maximal consecutive event sequences. By so doing it minimizes the number of moves from the falsified computer event sequence to the hypothesis computer event sequence.

There are two operations that are used frequently in the heuristic algorithm. One is to update the maximal consecutive event sequence in a temporary falsified computer event sequence $S_1$ and the other is to update the number of maximal consecutive computer event sequences in $S_1$. To do this we merely need to perform two major steps.

Assume that a maximal consecutive computer event sequence is being moved from location $i$ to lo-

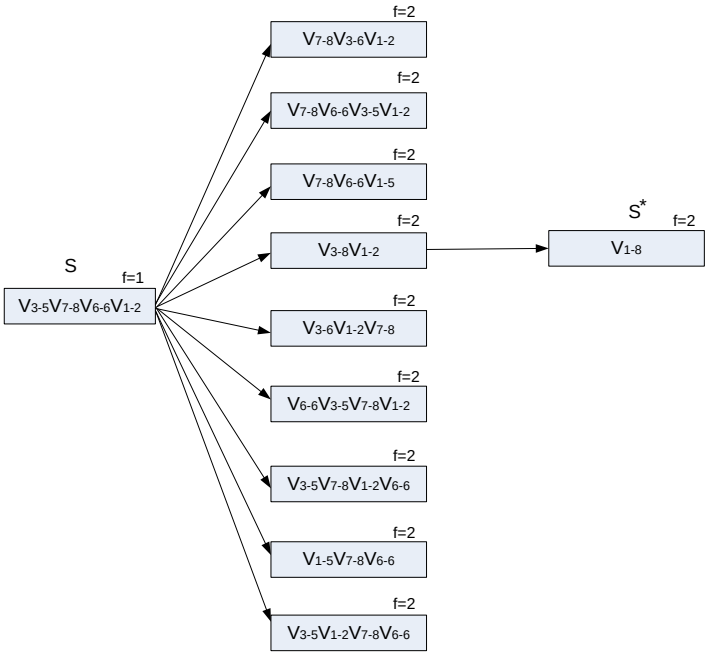


Figure 5: The $A^*$ search space after exploring $v_{3-8}v_{1-2}$

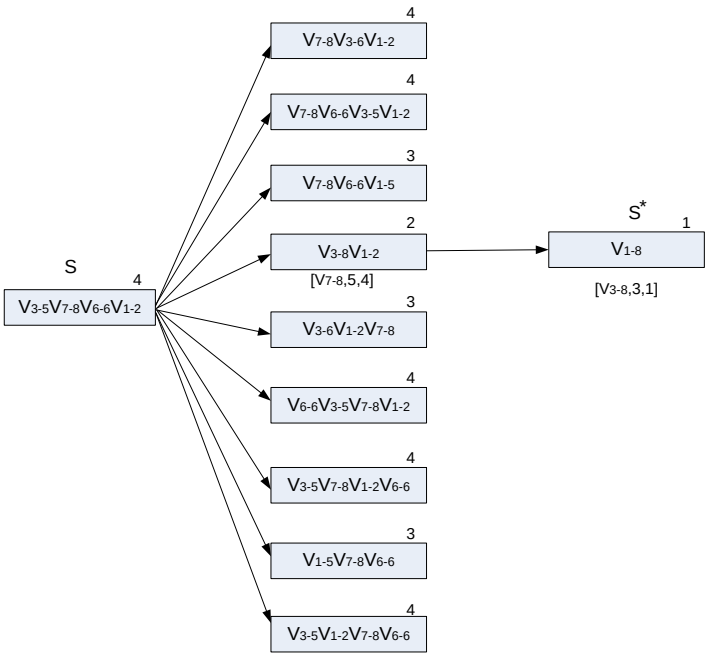


Figure 6: The search space of the heuristic algorithm

cation $j$ in $S_1$. We check if the maximal consecutive computer event sequences ending at locations $i-1$ and $i+1$ can be merged to form a new maximal consecutive computer event sequence, where $i+1 \leq m_1$ and $m_1$ is the number of maximal consecutive computer event sequences in $S_1$. This handles the situation where moving a maximal event sequence allows the sequences that surrounded it in its old location to be conjoined to create a new maximal consecutive sequence. If so, then they are merged to form a new maximal sequence, and we decrease the number of maximal consecutive computer event sequences by one.

Similarly, we further check whether or not moving a sequence to a new location allows it to be conjoined at one or both ends with its new neighbours to create an even longer maximal sequence. In other words, we want to know if the newly-moved sequence can be merged into a single maximal consecutive computer event sequences at both new locations $j-1$ and $j+1$, where $j < m_1$ and $m_1$ is the number of maximal consecutive computer event sequences in $S_1$. If so, we merge them and reduce the number of maximal consecutive computer event sequences by two. However, if the moved maximal sequence can be conjoined with its new neighbours at just one end, i.e., at location $j-1$ or $j+1$, we perform the merge and reduce the number of maximal consecutive computer event sequences by one.

Figure 7 shows an example of sequence concatena-

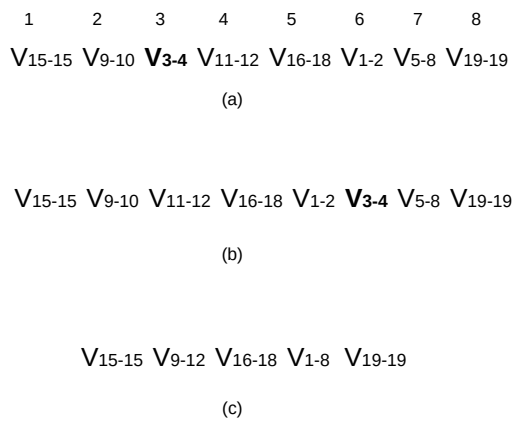


Figure 7: An example of sequence concatenation when moving a sequence

tion. In the example, maximal consecutive sequence $v_{3-4}$ is being moved from location 3 to location 7 in $S_1 = v_{15-15}v_{9-10}v_{3-4}v_{11-12}v_{16-18}v_{1-2}v_{5-8}v_{19-19}$. Figure 7(a) is the state of $S_1$ before the move; Figure 7(b) shows the intermediate state of $S_1$ after $v_{3-4}$ has been moved from location 3 to location 7 but before sequence concatenation is performed; Figure 7(c) displays the state of $S_1$ after sequence concatenation is done. Once sequence $v_{3-4}$ is moved from between them, maximal consecutive sequences $v_{9-10}$ and $v_{11-12}$ become adjacent in $S_1$ and are merged to form a longer maximal consecutive sequence $v_{9-12}$. Also when maximal consecutive sequence $v_{3-4}$ is placed between $v_{1-2}$ and $v_{5-8}$, all three are merged to form a new maximal consecutive sequence $v_{1-8}$. The total number of maximal consecutive sequences in $S_1$ is thus reduced by three in this case.

**Theorem 1** *Algorithm 5 can always find a feasible solution that moves maximal consecutive event sequences between* $\lceil (m-1)/3 \rceil$ *and* $m-1$ *times, where* $m$ *is the number of maximal consecutive event sequences in a falsified computer event sequence.*

*Proof.* Assume that a maximal consecutive computer event sequence is being moved from location $i$ to location $j$ in $S_1$. In the best case, when the maximal consecutive computer event sequences at the original locations $i-1$ and $i+1$ can be merged to form a new maximal consecutive computer event sequence and the maximal consecutive computer event sequence at new location $j-1$, the newly-moved sequence, and the event sequence at new location $j+1$ can all be merged to form a new maximal consecutive computer event sequence, the number of maximal consecutive computer event sequences in $S$ will be reduced by a total of three. Since there are $m$ maximal consecutive event sequences in $S$ initially, it may thus take as few as $\lceil (m-1)/3 \rceil$ moves to reduce the number of maximal consecutive event to one.

In the worst case, when the maximal consecutive computer event sequence being moved can be merged only with either the maximal consecutive computer event at location $j-1$ or the one at location $j+1$, the number of maximal consecutive computer event sequences will be reduced by one only. Since there are $m$ maximal consecutive event sequences in $S$ initially, it may thus take at most $m-1$ moves to reduce the number of maximal consecutive sequences to one.

In the following we use a simple example to illustrate how the heuristic algorithm works. The process is shown in Figure 6. In this example, $S^* = v_1v_2v_3v_4v_5v_6v_7v_8$ and $S = v_3v_4v_5v_7v_8v_6v_1v_2$. First of all, the heuristic algorithm transforms $S$ into a sequence of maximal consecutive computer event sequences $v_{3-5}v_{7-8}v_{6-6}v_{1-2}$. The number of maximal consecutive computer event sequences $m = 4$.

The heuristic algorithm explores all the sequences that can be obtained by moving one of the maximal consecutive sequences in $S$ from one location to another, to see which of them produces the minimal number of maximal consecutive computer event sequences. Since $v_{3-8}v_{1-2}$ is the one that has minimal number of maximal consecutive computer event sequences, in this case two, the search moves from $S = v_{3-5}v_{7-8}v_{6-6}v_{1-2}$ to $v_{3-8}v_{1-2}$.

The heuristic algorithm then similarly explores all the state-space neighbours of $v_{3-8}v_{1-2}$. In this case $v_{3-8}v_{1-2}$ has only one neighbour $v_{1-8}$ and the number of maximal consecutive computer event sequences in $v_{1-8}$ is one, so the search process terminates. Then the backtracking algorithm backtracks along the path from $S^*$ to $S$. As a result, the sequence of moves $[v_{3-8}, 3, 1]$ and $[v_{7-8}, 5, 4]$ is obtained.

In this particular case the heuristic algorithm finds the same two-move solution as the optimal one. Although this may not always be the case in general, the heuristic algorithm will always find a solution, and this will typically be one involving a small number of steps thanks to the process of conjoining maximal sequences into even longer ones whenever possible during an iteration of the algorithm. More importantly,

**Algorithm 5** A heuristic algorithm for the falsified computer event log reconstruction problem

**Require:** A falsified sequence of computer events $S$, and a hypothesised sequence of computer events $S^*$;
**Ensure:** A sequence of moves that transforms $S^*$ into $S$.
Transform $S$ into a sequence of maximal consecutive event sequences using Algorithm 1;
$m$ = the number of maximal consecutive events in $S$;
**for** $i = 1$, $m$ **do**
  $came\_from[i] = null$;
  $moves[i] = null$;
**end for**;
**while** $m > 1$ **do**
  $S_2 = S$;
  $m_2 = m$;
  **for** $i = 1, m$ **do**
    **for** $j = 1, m$ **do**
      $S_1 = S$;
      Move the $i^{\text{th}}$ maximal consecutive event sequence in $S_1$ from location $i$ to location $j$;
      Update the maximal consecutive computer event sequences in $S_1$ and the number of maximal consecutive event sequences in $S_1$, $m_1$;
      **if** $m_1 < m_2$ **then**
        $S_2 = S_1$;
        $m_2 = m_1$;
        $from = i$;
        $to = j$;
      **end if**
    **end for**
  **end for**
  $came\_from[S_2] = S$;
  $moves[S_2] = [S, from, to]$;
  $S = S_2$;
  $m = m_2$;
**end while**;
Backtrack the moves from $S^*$ to $S$ using Algorithm 4.

the heuristic algorithm can be applied to larger data sets than the optimal one, in general.

This is because the $A^*$ algorithm is a global optimal search algorithm. In the best case it may find a globally-optimal solution by exploring only a small part of the potential search space, but in the worst case it will be forced to explore all the alternatives in which an optimal solution may exist. By contrast, the heuristic algorithm is a local search algorithm. It explores only one local optimum area and converges to its local optimum (which may or may not be a global optimum). Therefore, even though it is possible for the $A^*$ algorithm to outperform the heuristic algorithm in particular cases, i.e., when the $A^*$ algorithm is lucky enough to find a globally-optimal solution in a small search area early, and the heuristic algorithm is forced to explore a large locally-optimal search space, this will not be true in general. For large data sets with multiple local optimums, the $A^*$ algorithm's average performance will be significantly worse than that of the heuristic algorithm.

## 5 Conclusion

Analysing computer forensic evidence is highly challenging. We have presented two algorithms for determining how a falsified computer log can be related to a hypothesised sequence of events, in terms of the effort required to transform one into the other. This gives us a sound basis for arguing about the likelihood of someone deliberately disguising their actions by tampering with computer logs, e.g., by adjusting the clock on their personal computer.

In future work we will perform empirical studies showing how the approach works in practice on actual large-scale computer logs. In addition, we will investigate how to construct the hypothesised computer event log using data from multiple trusted sources. In real life there may be many ways to obtain event logs, from timestamps on computer files, to logs on web servers, to logs of swipe card accesses on doors. A major unresolved challenge is to match an assumed "scenario" with all of these logs simultaneously.

## References


Berman, K. A. & Paul, J. L. (2005), *Algorithms: Sequential, Parallel and Distributed*, Thomson. ISBN 0-534-42057-5.

Best, P., Mohay, G. & Anderson, A. (2004), 'Machine-independent audit trail analysis—a decision support tool for continuous audit assurance', *Intelligent Systems in Accounting, Finance and Management* **12**(2), 85–102.

Boyd, C. & Forster, P. (2004), 'Time and date issues in forensic computing: A case study', *Digital Investigation* **1**, 18–23.

Carrier, B. (2002), Open source digital forensics tools: The legal argument, Technical report, @stake. `www.atstake.com/research/reports/acrobat/atstake_opensource_forensics.pdf`.

Case, A., Cristina, A., Marziale, L., Richard, G. & Roussev, V. (2008), 'FACE: Automated digital evidence discovery and correlation', *Digital Investigation* **5**, 65–75.

Chen, K., Clark, A., De Vel, O. & Mohay, G. (2003), ECF—event correlation for forensics, *in* 'Proceedings of the First Australian Computer, Network and Information Forensics Conference, Perth, 25 November'.

Gazagnaire, T. & Hélouët, L. (2007), Event correlation with boxed pomsets, *in* 'Proceedings of the 27th IFIP WG 6.1 international Conference on Formal Techniques For Networked and Distributed Systems (FORTE 2007)', Vol. 4574 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 160–17.

Gladyshev, P. & Patel, A. (2005), 'Formalising event time bounding in digital investigations', *International Journal of Digital Evidence* **4**(2).

Hart, P., Nilsson, N. & Raphael, B. (1968), 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Transactions on Systems Science and Cybernetics* **4**(2), 100–107.

Kent, K. & Souppaya, M. (2006), Guide to computer security log management, Technical report, U.S. Department of Commerce, National Institute of Standards and Technology. Special Publication 800-92.

Mohay, G. (2005), Technical challenges and directions for digital forensics, *in* 'Proceedings of the First International Workshop on Systematic Approaches to Digital Forensic Engineering', IEEE Computer Society Press, pp. 155–161.

Raghavan, S., Clark, A. & Mohay, G. (2009), FIA: An open forensic integration architecture for composing digital evidence, *in* 'Proceedings of the Second International Conference on Forensics in Telecommunications, Information and Multimedia, Adelaide, January 2009', pp. 83–94.

Richard III, G. G. & Roussev, V. (2006), 'Next-generation digital forensics', *Communications of the ACM* **48**(2), 76–80.

Schatz, B., Mohay, G. & Clark, A. (2004), Rich event representation for computer forensics, *in* 'Proceedings of the Asia Pacific Industrial Engineering and Management Systems Conference (APIEMS 2004), Gold Coast, Queensland, 12–15 December'.

Schatz, B., Mohay, G. & Clark, A. (2005), 'Generalising event forensics across multiple domains', *The Journal of Information Warfare* **4**(1), 69–79.

Schatz, B., Mohay, G. & Clark, A. (2006), 'A correlation method for establishing provenance of timestamps in digital evidence', *Digital Investigation* **3**, 98–107.

Willassen, S. Y. (2008*a*), Finding evidence of antedating in digital investigations, *in* 'Proceedings of the Third International Conference on Availility, Reliability and Security (ARES 2008), 4–7 March', pp. 26–32.

Willassen, S. Y. (2008*b*), Timestamp evidence correlation by model based clock hypothesis testing, *in* 'Proceedings of the First International Conference on Forensic Applications and Techniques in Telecommunications, Information and Multimedia, Adelaide, January 21–23'.

The *SC*-based PIR protocols assume that the *SC*'s secure memory can store a small number of data records. This assumption does not hold for multimedia databases, where records can be very long and storing even a single (and very long) record in *SC* can be a problem. Thus, task of designing a PIR protocol for databases with very long records is an interesting open problem.

**Acknowledgements**

Peishun Wang was supported by the RAACE scholarship, Macquarie University. Huaxiong Wang is supported in part by the Australian Research Council under ARC Discovery Project DP0665035 and the Singapore Ministry of Education under Research Grant T206B2204. Josef Pieprzyk was supported by the ARC grant DP0987734.

**References**

Ambainis, A. (1997), Upper bound on the communication complexity of private information retrieval, *in* 'Proc. of the 24th ICALP'.

Asonov, D., Freytag, J.-C. (4/2002), Almost optimal private information retrieval, *in* 'Proceedings of 2nd Workshop on Privacy Enhancing Technologies (PET2002)', San Francisco, USA.

Asonov, D., Freytag, J.-C. (5/2002), Private information retrieval, optimal for users and secure coprocessors, Technical Report HUB-IB-159, Humboldt University Berlin.

Beimel, A., Ishai, Y., Kushilevitz, E., Rayomnd, J.-F. (2002), Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval, *in* 'Proc. of the 43st IEEE Sym. on Found. of Comp. Sci.'

Chang, Y. (2004), Single Database Private Information Retrieval with Logarithmic Communication. *in* 'The 9th Australasian Conference on Information Security and Privacy (ACISP 2004)', LNCS, 3108, pp. 50-61, Springer-Verlag, Sydney, Australia.

Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M. (1998), Private information retrieval, *in* 'Journal of the ACM', 45. Earlier version in FOCS 95.

Chor, B., Gilboa, N. (1997), Computationally private information retrieval, *In* 'Proceedings of 29th STOC'.

Gentry, C., Ramzan, Z. (2005), Single-Database Private Information Retrieval with Constant Communication Rate, L. Caires et al. (Eds.): *in* 'ICALP 2005', LNCS 3580, pp.803–815.

Iliev, A., Smith, S. (2003), Privacy-enhanced credential services, *in* 'PKI Research Workshop', volume 2, Gaithersburg, MD. NIST.

Iliev, A., Smith, S. (2004), Private Information Storage with Logarithmic-space Secure Hardware, *in* 'Information Security Management, Education, and Privacy'. Kluwer, pp.201–216.

Iliev, A., Smith, S. (2005), Protecting Client Privacy with Trusted Computing at the Server, *in* 'IEEE Security & privacy'.

Sion, R., Carbunar, B. (2007), On the computational practicality of private information retrieval, *in* 'NDSS'.

Smith, S. W., Palmer, E. R., Weingart, S. H. (1998), Using a high-performance, programmable secure coprocessor, *in* 'Proceedings of the 2nd International Conference on Financial Cryptography'.

Smith, S. W., Safford, D. (2000), Practical private information retrieval with secure coprocessors. Technical report, IBM Research Division, T.J. Watson Research Center, 2000.

Smith, S. W., Safford, D. (2001), Practical server privacy with secure coprocessors, *in* 'IBM Systems Journal', 40(3).

Wang, S., Ding, X., Deng, R., and Bao, F. (2006), Private information retrieval using trusted hardware, *in* 'ESORICS', LNCS 4189, pp. 49-64.

Woodruff, D., Yekhanin, S. A. (2005), Geometric Approach to Information-theoretic Private Information Retrieval, *in* 'CCC', pp.275–284.

Yang, Y. Ding, X. Deng, R., Bao, F. (2008), An Efficient PIR Construction Using Trusted Hardware, *in* 'ISC', LNCS 5222, pp. 64–79.