Using Smart Cards for Tamper-Proof Timestamps on Untrusted Clients

Guenther Starnberger, Lorenz Froihofer and Karl M. Goeschka
Vienna University of Technology
Institute of Information Systems
Argentinierstrasse 8/184-1
1040 Vienna, Austria
{*guenther.starnberger, lorenz.froihofer, karl.goeschka*}*@tuwien.ac.at*

***Abstract*—Online auctions of governmental bonds and CO_2 certificates are challenged by high availability requirements in face of high peak loads around the auction deadline. Traditionally, these requirements are addressed by cluster solutions. However, with strong requirements regarding hardware ownership and only a few auctions per owner per year hardware clusters are a rather ineffective solution.**

Consequently, we contribute with a solution that alleviates the dependability problems by shifting them into the security domain: Key idea is to provide a secure timestamp service that allows users to place bids locally until the deadline, independent of server availability. This allows to mitigate peak-loads and network or server outages as the transfer of bids to the server can be delayed until after a performance peak or the repair of a failed component.

In this paper in particular, we contribute with a secure time synchronization and timestamping protocol tailored to online auctions where we apply secure timestamps on smart cards locally connected to the bidder's computer. Moreover, our timestamping protocol is robust with respect to man-in-the-middle delay attacks. Finally, we prove the feasibility of our approach based on a .NET smart card implementation and conclude with a discussion of current smart card limitations.

***Keywords*-Smart cards; Synchronization; Availability; Security;**

I. Introduction

First-price sealed-bid auction [1] scenarios generally exhibit high peak loads around the auction deadline as a majority of bidders tries to submit bids shortly before the deadline. Moreover, these auctions also exhibit high dependability requirements as an auction canceled due to technical reasons can lead to significant financial losses, because of market conditions changing over time. As a consequence, systems need to be designed for excessive workloads and high availability, leading to massive over-provisioning and high costs. However, for some governmental auctions, such as bond auctions and CO_2 certificate auctions, this over-provisioning is not cost efficient as auctions are only conducted a few times a year with the hardware being idle during the remaining time. Moreover, cloud computing solutions are not an option due to issues such as data ownership.

Key idea of our approach is to alleviate the dependability problems by shifting them into the security domain and by consequently solving the new security problems [2]. We increase dependability of the system by temporarily decoupling the individual components of the system from each other, and allowing users to place bids on trusted devices physically located at their place. However, by doing so we are decreasing the security of the system, as we are giving adversaries new options to attack the system. To address these issues we designed and implemented a secure timestamping approach that allows us to assign accurate timestamps to bids when they are placed and—in the case of high peak loads or temporary outages—queue the bids at the client and transfer them to the server at some later point in time.

One security problem with client-side timestamping is that software running on clients cannot be protected against attacks by malicious users. Manipulating the timestamps would enable a user to place bids even after the auction's deadline, thereby gaining potential advantages from information available after the deadline that may influence the user's decision on the bid.

By shifting the timestamping process from an untrusted computer to a secure device we can prevent users from tampering with critical parts of the software. However, only securing the software is not sufficient as users still retain full control over the network between the secure device and the auctioneer. As a consequence, malicious users can selectively delay packets and cause deliberate offsets during time synchronization. Our timestamping protocol mitigates these issues by enhancing on existing interval-based time synchronization techniques.

Our main contributions in this paper are a timestamping protocol and implementation capable of operation on secure devices and an examination of the restrictions placed by common smart card environments. Furthermore, we contribute with a threat model and propose techniques to further increase the security of such systems.

First, Section II discusses related work in academic and commercial areas to set the technological baseline. We

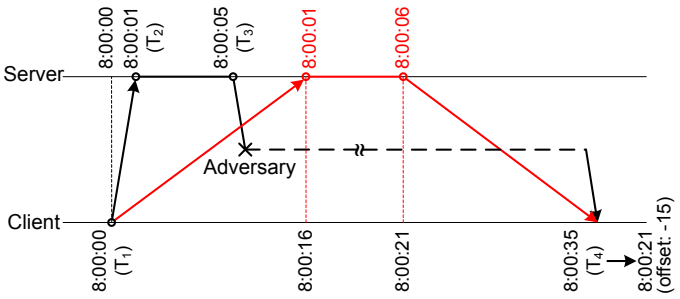

Figure 1. Asymmetric propagation delay

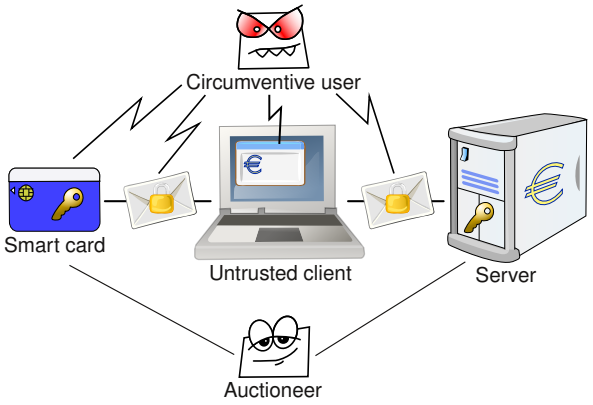

Figure 2. Application scenario

continue in Section III with an overview of our application scenario, our system architecture, our trust model, and an examination of delay attacks against time synchronization protocols. Based on this we contribute in Section IV with an interval-based time synchronization protocol prove the feasibility in Section V based on our smart card implementations. Finally, we draw our conclusions in Section VI.

II. Related work

While secure devices and time synchronization protocols are both well researched topics, the combination of using a secure time synchronization protocol in trusted hardware devices under physical control of potential adversaries has not been adequately researched in the academic area. This section introduces related time synchronization protocols that influenced the design of our smart card based time synchronization.

Network Time Protocol: The Network Time Protocol (NTP) [3] is a time synchronization protocol used on the Internet. To obtain timestamps from a remote server, NTP records four timestamps (T_1–T_4) as depicted in Figure 1. T_1 and T_4 are assigned according to the client's clock, while T_2 and T_3 are assigned according to the server's clock. NTP then uses the formula $\frac{(T_2-T_1)-(T_4-T_3)}{2}$ to determine the offset between the client's and the server's clock. Using the offset, it tries to estimate the skew of the local clock in relation to the server's clock and then gradually corrects the value and the frequency of the clock using a software-based phase-locked-loop (PLL) implementation.

NTP security analysis: A security analysis of NTP protocol version 2 was conducted in 1990 by M. Bishop [4]. While the evaluated NTP version is dated, the analyzed attacks, such as replay attacks, delay attacks, and denial-of-service attacks, are still relevant to today's time synchronization protocols.

Interval based time synchronization: Interval based time synchronization as proposed by Marzullo and Owicki [5] is an alternative view on time synchronization. Interval based protocols account for the fact that time synchronization with perfect precision and accuracy is not possible and therefore try to determine an interval that contains the real time, instead of a single point that represents the real time. An accuracy interval represents the fuzziness in setting, keeping and reading the time of a clock. For a perfect clock the length of such an accuracy interval is zero. In the real world, the effects of clock skew and network delays lead to an accuracy interval with a length larger than zero. With increasing time (without time synchronization) the size of the accuracy interval increases because of inaccuracies of the clock.

III. Application scenario

Our main application scenario are governmental auctions of CO_2 certificates and bonds with potentially up to thousands of bidders. These auctions use a variant of a first-price sealed-bid system [1] where the order of different bidder's bids does not matter and where bidders do not learn about bids placed by other bidders before the fixed auction deadline. There is no single winner, as the auctioned items, such as CO_2 certificates, are distributed among the best bids accordingly. Furthermore, bidders are allowed to submit updates to their bids until the auction deadline. Although unanticipated because of auction characteristics, experience in bond auctions has shown that most bidders place their bids shortly before the deadline, possibly due to psychological issues.

A. System architecture

Figure 2 depicts the individual components of our system architecture. The smart card is connected to the user's computer that relays messages between smart card and auctioneer. A circumventive user is able to attack different parts of the system. For example, physical attacks can be used against the smart card itself, delay attacks can be used against time synchronization messages relayed by the computer, and any software running on the computer may be manipulated.

- The *auction server* is responsible for hosting the auction Web application and additionally acts as a time server. Bids timestamped by a smart card are relayed over the untrusted client to the auction server.

- The *smart card* timestamps bids provided by the untrusted client with the current time. It obtains the time using a time synchronization protocol. Messages between smart cards and time servers are relayed by the untrusted client in between. Messages between smart cards and bidders are secured with QR-TANs [6].
- The *untrusted client* enables communication between smart cards and auctioneer. Messages transmitted between a smart card and an untrusted client are encoded as APDUs (*Application Protocol Data Unit*), while the untrusted client uses standard Internet protocols, such as TCP (*Transmission Control Protocol*) and UDP (*User Datagram Protocol*), to communicate with the auction server. Only non-security-critical software operations are executed on the untrusted client.

To provide correct time on trusted devices, such as smart cards, we employ a secure time synchronization protocol. The given reference time is obtained from servers controlled by the auctioneer.

B. Attacks against time synchronization

To motivate the problem, Figure 1 shows an example of how an adversary can use asymmetric propagation delays to affect time synchronization when a time synchronization protocol assumes symmetric propagation delays and uses the offset formula used by NTP (described in Section II). Initially, client and server are perfectly synchronized. The client sends its request at 8:00:00, which is received by the server at 8:00:01. The server processes the request and sends the response at 8:00:05. On the way back to the client, the response is delayed by an adversary, causing the client to receive the delayed response at 8:00:36. When the client calculates the offset to the server, this yields a value of -15 seconds, implying the time 8:00:36 on the client's clock corresponds to the time 8:00:21 at the server's clock. Therefore, the adversary delaying the message on the way back to the client eventually delays the client's clock by 15 seconds.

Lundelius and Lynch proved that the optimum bound achievable when synchronizing n clocks is $u \times \left(1 - \frac{1}{n}\right)$, where u is the uncertainty in message transmission time and n is the number of hosts [7]. In a traditional client-server setup where a single client obtains its time from a single server, the maximum message transmission delay between client and server limits the uncertainty, and thereby the possibilities of an attacker to $\frac{u}{2}$. In terms of the four timestamps T_1–T_4 we can define the maximum error as $\frac{(T_4-T_1)-(T_3-T_2)}{2}$.

C. Trust model

Our application scenario is fundamentally different from traditional time synchronization and timestamping scenarios. In traditional time synchronization protocols, individuals interested in obtaining accurate timestamps have full control over clients. On the other hand, servers are often only reachable over an untrusted network. In some cases, such as public *Network Time Protocol* (*NTP*) servers, individual servers cannot be fully trusted.

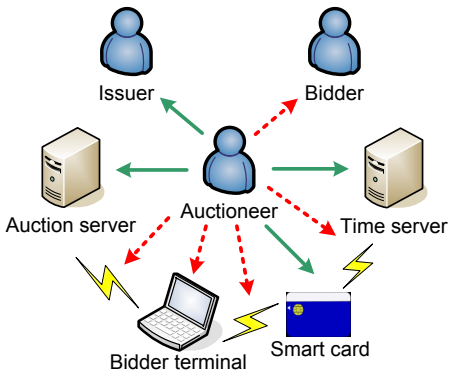

Figure 3. Trust model

In our scenario, the auctioneer has full control over the time servers, but no physical control over the clients. Successfully tampering with smart cards may allow malicious users to issue bids even after the deadline has already passed. Therefore, it must be guaranteed that the underlying hardware is sufficiently secure against such types of attacks.

Figure 3 shows the trust relationships in our application scenario. Green solid arrows indicate that the respective role is able to reasonably trust another role or component, if reasonable security mechanisms, e.g., encryption of network links or installed firewall and antivirus scanner, are applied or contracts between different roles are arranged. Otherwise, the untrusted relationship is indicated via a red dashed arrow.

Based upon existing technology, the auctioneer is able to trust the own infrastructure (auction server, network links, and smart card), the issuer, and the time server infrastructure. Although, we assume the auctioneer may trust the bidder in general, she cannot fully trust the bidder as the manipulation of the clock on the bidder's terminal is undetectable. Consequently, the auctioneer has to distrust the bidder with this respect. Similarly, the auctioneer has to distrust the bidder's terminal and the communication channel between terminal and smart card.

The communication between the smart card and the time server can be trusted with respect to message integrity, but as communication is only performed via the bidder's terminal, communication delays might be introduced by the bidder to influence the time synchronization algorithm. Therefore, this establishes a source of distrust for smart card - timeserver communication. Similar considerations apply to the communication link between bidder terminal and auction server as the bidder might interrupt the network connection to pretend a network failure.

IV. Time synchronization and timestamps

For time synchronization and timestamping we are using an interval based approach based on Marzullo's work [8], allowing us to represent time as a correctness interval that represents the inaccuracies due to network delays and clock characteristics, such as clock skew and limited precision. An interval representing a particular time is correct, if the represented time is within the bounds of the interval. Two intervals are *compatible* if they represent the same time value, such as "5 pm" or "now". Two intervals are *consistent* if their intersection is not zero. Thus, in the case where two compatible intervals are not consistent, at least one of these intervals must be incorrect. In our algorithm we use *offset intervals* that represent the difference between the smart card's local clock value and the values of the two intervals endpoints.

Both, our time synchronization and timestamping algorithms are adapted to smart card environments that may exhibit high propagation delays between smart card and timeserver. In particular, our contributions over the state of the art are:

- An adaption of Marzullo's interval approach to guarantee that each time synchronization step—independent of the propagation delay—only increases, but never decreases, the accuracy of the local clock.
- An adaption of Marzullo's intersection approach to a single time server scenario, allowing us to detect if an attacker has been able to successfully tamper with the time on the smart card.
- A roundtrip delay mitigation approach that allows use to decrease the size of intervals applied as timestamps, if there are confirmed problems at the auctioneers infrastructure that lead to consistent, measurable delays in the processing of time stamps.

The definitions used in this section are based on *Network Time Protocol Version 4 – Reference and Implementation Guide* [9] by David L. Mills and on *Maintaining the time in a distributed system* [8] by Marzullo and Owicki.

A. Time synchronization

We define our time synchronization protocol based on algorithms implementing the different functions: Algorithm 1 illustrates the main control loop in the client. The time transfer from the server to the client is done by Algorithm 2 with Algorithm 3 being responsible for increasing the interval's size due to clock skew. Both algorithms work with offset intervals that represent the uncertainty in clock synchronization due to clock skew and network delays, for example. As these offset intervals do not represent a real point in time, Algorithm 4 converts the offset intervals to absolute time intervals that are then cryptographically signed in Algorithm 5. The interpretation of the timestamped intervals is the server's task and illustrated in Algorithm 6.

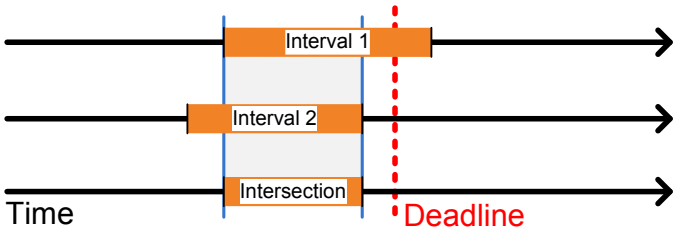

Figure 4. Interval overlapping with deadline and intersection between two intervals

1) Time synchronization protocol: Algorithm 1 shows the time synchronization algorithm running on the client. First, the algorithm initializes the offset interval i_{local} to $[-\infty, \infty]$—as we do not have any information about the current time yet. It proceeds, by obtaining an interval from the timeserver that is subsequently stored in i_{server}. The local clock value returned by $get\_time()$ when the time synchronization step was started is stored in t_{last_set}.

If the two compatible intervals i_{local} and i_{server} representing the current time do not overlap, we can conclude that one of these intervals must be incorrect. In either case this represents a fatal error and we disable the smart card, requiring the bidder to use other means to submit bids. As the inverse condition does not need to be true (two compatible intervals may intersect if one or both of these intervals are incorrect), this test does not detect all types of failures. However, the maximum error in these cases can be considered rather low, as it cannot be larger then the smallest value of i_{server} obtained during all preceding time synchronization steps.

When assuming that both i_{local} and i_{server} are correct, we can follow that the *intersection* of i_{local} and i_{server} will also represent the correct time. Therefore, we replace i_{local} by the *intersection* of i_{local} and i_{server} to decrease the size of the resulting interval and to more accurately represent the current time. An example is given in Figure 4.

Finally, the algorithm waits for the next time synchronization step while allowing the smart card to timestamp bids in between. Before executing the next round, we increase the size of i_{local} with the $extend()$ function, to account for inaccuracies due to clock skew.

2) Time transfer from server to client: The $recv\_interval()$ function given in Algorithm 2 is used to obtain the time from a remote server: After receiving the response by the server, the smart card first calculates the server's synchronization distance (Γ) from the server's root dispersion (E) and the server's root delay (Δ), which are both part of the NTP response message [3]. The synchronization distance represents the maximum error in the server's response due to all causes. Afterwards two offsets are calculated, by decreasing and increasing the obtained offsets by half the roundtrip delay of the time synchronization request. Finally, the offset values are

Algorithm 1 Obtaining time from server

$i_{local} \leftarrow [-\infty, \infty]$
loop
  $t_{last\_set} \leftarrow get\_time()$
  $i_{server} \leftarrow recv\_interval()$
  if $intersection(i_{local}, i_{server}) == 0$ **then**
    $abort\_with\_error()$
  end if
  $i_{local} = intersection(i_{local}, i_{server})$
  while $wait\_for\_timeout()$ **do**
    {// Apply timestamps on incoming bids}
  end while
  $i_{local} = extend(i_{local})$
end loop

combined with the synchronization distance and returned as a single offset interval.

Algorithm 2 $recv\_interval()$: Time synchronization request

$response \leftarrow receive\_response\_from\_server()$
$\Gamma \leftarrow response.E + \frac{response.\Delta}{2}$
$o_1 \leftarrow (T_3 - T_4)$ {// $(T_3 - T_4) == offset - \frac{roundtrip\_time}{2}$}
$o_2 \leftarrow (T_2 - T_1)$ {// $(T_2 - T_1) == offset + \frac{roundtrip\_time}{2}$}
$i_{server} \leftarrow [o_1 - \Gamma, o_2 + \Gamma]$
return i_{server}

3) Clock skew: The $extend()$ function in Algorithm 3 increases an interval to account for the effects of clock skew. ρ_K represents the precision of the remote clock and is included as part of the time server's response. ρ represents the precision of the local clock and is estimated by the auctioneer. $\Phi(X)$ represents the maximum error due to clock skew during a period of length X. In our application, we are interested in the clock skew since the last time synchronization step t_{last_set}. $extend()$ combines the different effects of these components, and increases the size of the interval accordingly.

Algorithm 3 $extend(i)$: Transform offset interval due to effects such as clock skew

$\varepsilon \leftarrow \rho_K + \rho + \Phi(get\_time() - t_{last\_set})$
$i_{new} \leftarrow [i.left\_offset - \varepsilon, i.right\_offset + \varepsilon]$
return i_{new}

B. Timestamping

1) Timestamping: When timestamping a bid we first convert the offset intervals i_{local} and i_{server} to absolute intervals with the approach shown in Algorithm 4. Afterwards, we append these absolute intervals plus a sequence number to the bid, and sign the resulting data with a digital signature as shown in Algorithm 5.

Algorithm 4 $abstime(i)$: Transform offset interval to *current* time

$t_{current} \leftarrow get\_time()$
$i \leftarrow extend(i)$
$i_{abs} \leftarrow [t_{current} + i.left\_offset, t_{current} + i.right\_offset]$
return i_{abs}

Algorithm 5 $timestamp(bid, i_{local}, i_{server}, t_{local})$: Timestamp incoming bid

$seq \leftarrow seq + 1$
$ts_{local} \leftarrow abstime(i_{local})$
$ts_{server} \leftarrow abstime(i_{server})$
$data \leftarrow concat(bid, seq, ts_{local}, ts_{server}, get\_time())$
$sign(data)$

While it would be sufficient to only include the absolute interval derived from i_{local} in the timestamp, the interval derived from i_{server} allows the auctioneer the mitigation of high peak loads as described later on.

A sequence number is required to verify that all bids signed by the smart card have been received at the server. After the auction ended, we therefore require the smart card to transfer the value of its sequence number to the server.

2) Server-side timestamp interpretation: Algorithm 6 shows how timestamps are interpreted at the server. First, it is checked if there have been confirmed problems at the auctioneer during all of the time represented by the timestamp. If this is the case, we can use the $adapt()$ function, to decrease the size of ts_{server} accordingly.

As example, consider that the auctioneer's monitoring infrastructure confirms that there has been an additional network delay of 10 seconds for messages that travel from the network to the auctioneer. This would result in the right hand side of each interval obtained over the network during this delay—given as $(T_2 - T_1)$—to be increased by 10 seconds. Therefore, our $adapt()$ function can shift the right hand side of the interval 10 seconds to the left, to mitigate for the increased propagation delay.

Finally, the algorithm calculates the actual timestamp by building the intersection of ts_{local} and ts_{server}. While under normal conditions ts_{local} is already a subset or equal to ts_{server}, in cases where ts_{server} was recalculated with $adapt()$, the intersection potentially allows to reduce the size of the interval.

Algorithm 6 Timestamp processing at server

if *signature incorrect* **then**
  $abort\_with\_error()$
else if $ts_{server} \subseteq t_{confirmed\_problems\_at\_auctioneer}$ **then**
  $ts_{server} \leftarrow adapt(ts_{server})$
end if
$ts_{timestamp} \leftarrow intersection(ts_{local}, ts_{server})$

3) Decision if a bid should be accepted: Our application scenario requires that any bid timestamped before the auction deadline must be accepted, while any bid timestamped after the deadline must be rejected. In cases where both interval endpoints are either before or after the deadline, the decision is clear. However, in cases where the interval overlaps with the deadline, the auctioneer cannot fully assert, if a bid was placed before, or after the deadline.

In most cases, intervals of timestamps are relatively small. The only case when a large interval will be assigned is if there has not been any single time synchronization step with a reasonable roundtrip delay and—additionally—if the auctioneer did not detect any overload at his local network, and thus does not call $adapt()$ on the assigned timestamp. In such cases it can be possible that large intervals are caused by delay attacks caused by malicious bidders.

To mitigate for potential attacks, we are always comparing the right hand side of an interval—which represents the latest possible time of a bid—to the deadline. In cases where intervals are small, it does not matter which part of the interval is compared with the deadline, as the expected interval size is at most a few hundred milliseconds. However, in cases of large intervals—e.g. due to delay attacks—comparing the right hand side to the deadline prevents these large intervals from delaying the auction deadline for a particular user.

The worst case from the bidders' perspective occurs if every single time synchronization step features high propagation delays to the server, while there are no confirmed problems at the auctioneers infrastructure. In such cases, the deadline comparison with the interval's right endpoint advances the bidders' deadline, and thereby requires bidders to submit bids sometime before the auction deadline. However, as bidders can always query the smart card's time, they can detect such cases in advance, allowing them to also use other mitigation strategies, such as switching to a different network connection.

Figure 5 shows how this approach affects the accuracy of time stamps with measurements based on our protocol implementation. The example simulates a delay attack with high propagation delays for responses sent from the server. The "reference clock" is regarded as the server's time, while the "offset against reference clock" is the right hand endpoint of the interval stored on the smart card. In the example the effects of clock skew are negligible and therefore not visible.

Initially, the "offset against reference clock" is equal to the propagation delay of the first time synchronization message sent from the client to the server, as the formula for the right endpoint $(T_2 - T_1)$ is only dependent on the propagation delay to the server, but not on the propagation delay of responses sent by the server.

Subsequent time synchronization steps will calculate the intersection of the smart card's local offset interval and the offset interval obtained from the time server. Therefore, with each time synchronization step the right endpoint can either stay equal or move to the left, but never to the right. This behavior can be seen in Figure 5, as the "offset against reference clock" only decreases but never increases as clock skew is negligible.

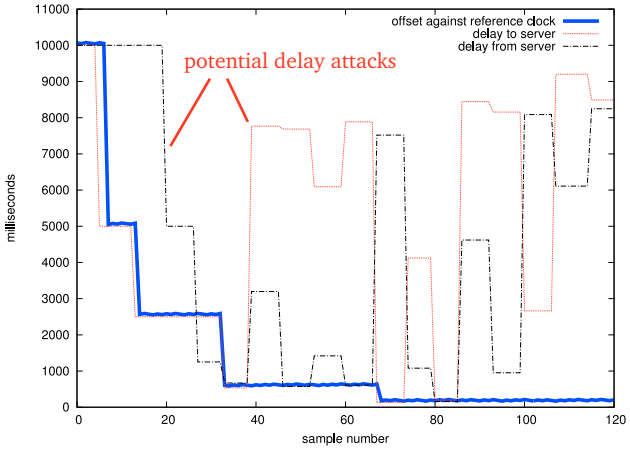

Figure 5. Response to delay attack

In cases where the smart card's clock skew would be noticeable, the "offset against reference clock" would slowly increase over time, but still be reset by each time synchronization step, where the propagation delay to the server is lower than the offset.

V. Smart Card Implementation

In this section we examine techniques to implement our approach on .NET cards and Java cards. While our general approach is also applicable to other types of secure devices, such as TPMs (*Trusted Platform Modules*), our original application scenario, and thereby also our evaluation, targets smart card environments.

To implement our time synchronization protocol on a smart card, the smart card needs to possess the following capabilities:

1) Access to an internal timer clocked by an oscillator internal to the smart card. The frequency of the oscillator must not depend on the frequency provided by the card reader to the `CLK` contact.
2) A means to detect if the counter has been reset, e.g., because the power supply to the smart card was interrupted.
3) The ability to detect overflows of the counter. However, as explained later, a software based workaround is possible, if this feature is missing.

A read-only timer is sufficient, as the difference between the reference time and the timer's value can be stored as a variable on the smart card. Furthermore, we do not presume a battery powered clock that allows keeping the time while the secure device is not connected to a computer, as this feature is not available on a majority of examined devices.

The NTP packet format is a potential cause for compatibility issues as smart cards need to successfully generate and parse the individual fields of packets. This parsing cannot be outsourced to external applications as smart cards must be able to verify signatures applied by the time server. If the values would be extracted by outside applications, smart cards would have no means of verifying that the extracted values actually correspond to the cryptographically signed response of an NTP server.

NTP messages are parsed and generated on-card. The smart card communicates via APDUs with a client application running on the local PC, relaying NTP messages between smart card and Internet. We use MD5 based signatures as described in [10] with an individual key for each smart card, requiring smart cards to provide an MD5 hash function to on-card applications. While collisions in MD5 have been found [11], control over both, NTP client and server implementations, allows us to increase the security beyond the NTP specification by replacing MD5 with any alternative hash function.

For the evaluation discussed below we implemented our protocol on a *Gemalto .NET IM V2* .NET smart card and—additionally–tested the capabilities of a technology preview release (simulator) of the connected edition for Java Card 3 and evaluated the limitations of Java Card 2 based on a *NXP JCOP 41 V2.2.1/72K* card. All three types of cards are programmable and allow the installation of appropriate software.

A. .NET card

The .NET smart card API provides access to the 32-bit read-only `System.Environment.TickCount` property that returns the the number of milliseconds passed since the smart card has been powered on. The timer within .NET cards runs continuously for 24.9 days until it reaches `Int32.MaxValue`. Afterwards, it wraps back to zero. According to the API, the resolution of the `TickCount` property is at least 500 milliseconds. Similar to `System.currentTimeMillis()` in Java Card 3, the `TickCount` property can be used to measure the time between two invocations.

The `TickCount` property overflows after 24.9 days, making it impossible for the software running on the smart card to assess whether e.g. 25.0 days or 0.1 days have passed. This is mostly a theoretical issue, as the usual duration of auctions is much shorter in our application scenario. Detection of overflows would be possible if the software running on the smart card could check the value of the `TickCount` property in intervals smaller than 24.9 days. However, the .NET API does not allow for the execution of background threads. Hence, code execution depends on external APDUs transmitted to the smart card. As these APDUs need to be transmitted over the bidders computer, it cannot be guaranteed that the bidder abides by a policy of sending APDUs with intervals smaller than 24.9 days.

To work around this problem, the auctioneer can limit the maximum time allowed between the application of a timestamp and the transmission to the auction system to an interval smaller than the maximum value that can be represented by the timer. While this does not prevent the timer from overflowing, it allows the auction system to detect if an overflow occurred.

Concluding, the .NET card fulfills our requirements with the overflow restriction described above.

B. Java Card 3

In Java Card 3 the *Application Programming Interface for the Java Card Platform, Connected Edition* provides a `System.currentTimeMillis()` method that returns the current time in milliseconds. While the Java Card API also provides a method `synchronizeTime()` that allows obtaining the time from an external time reference, the API does not specify what external time reference is used by implementations. Therefore, we cannot trust that the time obtained by a Java Card 3 is valid, as the external time reference can be under control of an adversary, e.g., if the time is obtained from an external card reader. As a consequence, our only assumption is that the difference in the responses of two `System.currentTimeMillis()` invocation reflects the interval in milliseconds between these invocations.

A further complication is that some future Java Card 3 implementations might automatically obtain the time from an external reference. In order to detect such events, Java Card 3 allows to register an `PlatformEvent.EVENT_CLOCK_RESYNCED_URI` event handler able to detect `event:///platform/clock/resynced` events. Such events are fired after the Java card obtained its time from an external reference. The time delta between the old time and the new time is included in the arguments of the event handler. Capturing the events allows a time synchronization protocol to mitigate the offsets caused by such synchronization steps.

While Java Card version 3 has been standardized, there are currently no cards available on the market. Consequently, Java Card 3 only represents an option for the future.

C. Java Card 2

The official Java Card 2 API does not include methods to access the timers of smart cards. However, there are implementations of Java Card 2 compatible devices that include access to an internal clock, such as the DS1955B and DS1957B cryptographic iButtons produced by Dallas Semiconductor (now a subsidiary of Maxim Integrated Products). The internal clock of these devices is quartz-driven and powered by an internal battery. However, according to information we received by a Maxim engineer, Java-powered iButtons have been discontinued.

As a consequence, we based our Java Card 2 prototype version on standard Java cards. While the API of these cards

does not provide access to a timer, we used a software based timer simulation for evaluation purposes, by regularly updating the value of an internal timer variable with incoming APDUs and by providing a corresponding access function that guarantees a strictly monotonic increasing behavior.

In our implementation it was not possible to provide a `System.currentTimeMillis()` compatible interface, as `currentTimeMillis()` returns a 64-bit *long* value, while the largest data type supported by Java Card 2 is (depending on the card) either signed 16-bit or signed 32-bit. However, Java Card version 2.2.2 provides a `BigNumber` class that allows using multiple smaller data types to simulate a larger data type. As, we could not obtain Java cards implementing the relatively recent Java Card 2.2.2 specification, we implemented our own `BigNumber`-like class based on a subset of the `BigInteger` class in Java SE.

In a first preliminary performance evaluation we tested a simple application that sums all integers between 10000 and 20000. On a JCOP card this test shows about 200 operations per second. In comparison, a .NET card natively implementing the required data types is able to execute about 5800 operations per second, nearly a thirty-fold difference.

To sum up, Java cards version 2 are not adequate for time synchronization. They miss standardized access to a timer as well as appropriate data types to process time synchronization packets.

VI. Conclusion

In this paper we contributed with a secure time synchronization and timestamping approach for online auctions. These auctions are characterized by high peak loads near the end of auctions as well as high dependability requirements. Traditionally, these requirements are met by geographically distributed server clusters, leading to massive over-provisioning and massive costs.

With the application of secure timestamps at the bidder's site, we can mitigate peak loads as well as node or link failures as the transfer of timestamped bids can be delayed until after a performance peak or the repair of a failed component. However, simply implementing timestamping in client-side software is not feasible as it is not possible to effectively secure such software against attacks by circumventive bidders. Consequently, we contributed with a secure time synchronization and timestamping protocol for resource-constrained devices to enable secure timestamps to be applied within smart cards, for example. Moreover, we showed the feasibility of our approach based on a .NET smart card implementation.

We are convinced that our approach contributes to increased dependability and more efficient resource usage in distributed applications able to benefit from a distributed secure timestamping facility. While our application scenarios cover online auctions of government bonds and CO_2 certificates, we aim at broader application of our solution approach in future work.

Acknowledgments

The authors would like to thank Juraj Holtak for the Java Card and .NET implementations of our time synchronization protocol. This work has been partially funded by the Austrian Federal Ministry of Transport, Innovation and Technology under the FIT-IT project TRADE (Trustworthy Adaptive Quality Balancing through Temporal Decoupling, contract 816143, http://www.dedisys.org/trade/).

References

[1] R. P. McAfee and J. McMillan, "Auctions and bidding," *Journal of Economic Literature*, vol. 25, no. 2, pp. 699–738, June 1987. [Online]. Available: http://www.jstor.org/pss/2726107

[2] L. Froihofer and K. M. Goeschka, "Balancing of dependability and security in online auctions," in *Dependable Systems and Networks, 2008. DSN '08. International Conference on*, 2008.

[3] D. Mills, "Network Time Protocol (Version 3) Specification, Implementation and Analysis," RFC 1305 (Draft Standard), Mar. 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1305.txt

[4] M. Bishop, "A security analysis of the NTP protocol version 2," in *Computer Security Applications Conference, 1990., Proceedings of the Sixth Annual*, Tucson, AZ, USA, Dec. 1990, pp. 20–29.

[5] K. Marzullo and S. Owicki, "Maintaining the time in a distributed system," in *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983, pp. 295–305.

[6] G. Starnberger, L. Froihofer, and K. M. Goeschka, "QR-TAN: Secure mobile transaction authentication," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, Fukuoka, Mar. 2009, pp. 578–583. [Online]. Available: http://dx.doi.org/10.1109/ARES.2009.96

[7] J. Lundelius and N. A. Lynch, "An upper and lower bound for clock synchronization," *Information and Control*, vol. 62, no. 2/3, pp. 190–204, 1984.

[8] K. Marzullo and S. Owicki, "Maintaining the time in a distributed system," *SIGPS Oper. Syst. Rev.*, vol. 19, no. 3, pp. 44–54, 1985.

[9] D. L. Mills, "Network time protocol version 4 reference and implementation guide," University of Delaware, Tech. Rep. 06-6-1, June 2006. [Online]. Available: http://www.eecis.udel.edu/~mills/database/reports/ntp4/ntp4.pdf

[10] ——, *Computer Network Time Synchronization: The Network Time Protocol*. Boca Raton, FL, USA: CRC Press, Inc., 2006.

[11] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *EUROCRYPT*, ser. LNCS, R. Cramer, Ed., vol. 3494. Springer, 2005, pp. 19–35. [Online]. Available: http://dx.doi.org/10.1007/11426639_2