

# Secure Client-Side-Only Multiplayer Gaming API

(aka The Jury Game)

YOAV ZIBIN

Come2Play

yoav@come2play.com

September 24, 2008

## Abstract

Multiplayer gaming platforms (such as Come2Play, Skype, Nonoba, Oberon) offer game developers an API to develop new games. Having a secure API is critical to prevent hackers from unlawfully winning a game. Until today, to have a secure API, a developer had to write a *server-side* extension that determines the *game outcome*. However, a server-side extension is cumbersome to write (because you have to master two programming languages: for the client- and server-side), error-prone, hard to debug, and risky for the gaming platform that runs third party code on its servers.

This paper presents the first *Secure client-side-only API* (for short *SecureAPI*), i.e., the API is secure (the game outcome cannot be changed by hackers) and the API uses only client-side code (without any server-side extensions). SecureAPI mimics real-life games in which each player verifies that other players follow the game rules. In case of disagreement among the players, the server convenes a *jury* that finds the hacker. Using SecureAPI, one can develop secure multiplayer games using only client-side code, without using any server-side extensions.

SecureAPI is an open-source standard developed by *Come2Play*, with an open-source flash emulator<sup>1</sup>. Come2Play freely hosts 3<sup>rd</sup> party flash games and shares advertising revenues with the game developers.

## 1 Introduction

Multiplayer casual gaming is a growing and diverse field, ranging from dedicated gaming portals (such as Pogo, Yahoo and MSN zone), widgets and gadgets (iGoogle), instant-messengers (Skype, ICQ, Meebo), and gaming in social networks (Facebook, MySpace, Orkut). Various gaming platforms (such as Come2Play, Nonoba, Oberon, MSN, Yahoo, Skype) offer game developers

---

<sup>1</sup><http://code.google.com/p/multiplayer-api>

a *multiplayer gaming API* that allows the creation of new games. The gaming platforms profit from new games that increase user traffic, and the game developers profit from new distribution channels and sometimes sharing in either revenues (e.g., Come2Play and Skype) or in the advertising space (e.g., Nonoba). The game developer can focus solely on making the game, without worrying about the surrounding addictive features that are provided by the gaming platform, such as a chat area for the players, best scores management, inviting friends, managing saved games, token or rating system, etc. The gaming platform provides a container for the game that implements the above features and handles the network communication with the platform servers. The players communicate with each other either via the platform servers or directly in a peer-to-peer communication. The game outcome is recorded on the platform servers that maintains a database including gaming history, players' statistics, and a rating/token/reward system to increase game addictiveness.

The API contains functions to pass data between a player and a gaming server, or directly between the players, and a way to decide on the game outcome (e.g., who is the winner or what is the score of each player). Most players will use the API lawfully and report the correct game outcome to the server. However, there will always be a minority of *hackers* that will try to change the game outcome in their favor. We say that an API is *secure* if the server always stores the correct game outcome, i.e., *hackers cannot unlawfully change the game outcome*. Having a secure API is of course critical when the players bet with real money. Even when the stakes are virtual (e.g., in a token or rating system), a company will lose users if hackers can crack their API.

This paper presents the first *Secure client-side-only API* (for short *SecureAPI*), i.e., new games do not require any server-side extensions. SecureAPI uses a novel approach that mimics real-life games in which each player verifies that other players follow the game rules. In case of disagreement among the players, the server randomly selects a *jury* that finds the hacker.

SecureAPI is an open-source standard developed by Come2Play, with an open-source flash emulator<sup>2</sup>. Come2Play<sup>3</sup> developed a container for flash games that use SecureAPI. Come2Play distributes the games and shares advertising revenues with game developers.

**Outline** Section 2 reviews previous work on gaming APIs. SecureAPI is presented in Section 3, with examples such as TicTacToe, Snake, Backgammon, Battleship, Poker, and more in Section 4. The server implementation is given in Section 5. Finally, the paper concludes in Section 6.

## 2 Previous Work

Gaming APIs today can be divided into two categories:

---

<sup>2</sup><http://code.google.com/p/multiplayer-api/>

<sup>3</sup><http://come2play.com>

**Insecure client-side API** The game logic runs purely on the clients that play the game, and the game outcome is determined solely by the clients that send it to the gaming server. In other words, there is no specific server-side logic per game. Therefore this kind of API is inherently insecure, because a hacker can send the server an arbitrary game outcome. Even if the client code and network messages are encrypted or obfuscated, hackers may crack such protections and change the game outcome.

This API is used mainly due to its simplicity for the game developer and the gaming platform (such as in Skype and MSN instant messengers). Because the players are usually acquainted with one another, it is less likely that one of them will wish to hack the game and change its outcome. In games with a secret state (e.g., Battleship or Poker), a hacker can either fake his secret state or know the secret state of other players. Hackers invest a lot of time in order to change the game outcome by using various techniques ranging from changing the frame-rate (speed) of games, examining memory contents, or even faking network messages.

**Secure client- and server-side API** A game that uses this API has two parts: client-side code that handles the game graphics, and server-side code that handles the game logic and determines the game outcome. This API is secure, however the gaming company takes a huge risk when its servers run code written by a third party (even if it runs within a very restrictive sandbox).

Nonoba offers such an API, where client-side code is written in ActionScript (for flash), and server-side code is written for the .Net platform. A game is submitted by sending Nonoba two files: one for the client-side and another for the server-side. Nonoba warns the game developers that their server-side logic should use a small amount of memory, CPU, and network traffic per player.

In addition to the dangers of running server-side code, this approach has disadvantages to the game developer as well, mainly due to its complication, and issues regarding scalability, fault-tolerance, and synchronization. Writing server-side code is a delicate and complicated task that requires understanding of the system as a whole, and following a strict set of rules to ensure deadlock freedom, tolerance to server faults, and migration of code and data to obtain scalability. It is also complicated to debug a running application, log errors, and profile the runtime, memory or network footprint of the game. Moreover, using two programming languages (for the client- and server-side), with a different programming model, is error-prone and leads to code duplication.

For example, consider a flash game of pool that uses a physical simulation to move the balls on the table. Collision detection might use flash's built-in methods, such as

```
DisplayObject.hitTestObject(otherDisplayObj:DisplayObject)
```

When writing the server-side logic for this pool game, the game developer would have to write a .Net equivalent for this flash method.

The next section describes a secure API that uses only client-side code. Secure API is applicable to more than just a multiplayer gaming API; it is applicable to any client-server architecture where the clients follow *different protocols* and invoke a *sensitive server operation*. In the gaming API, the protocols are games and the sensitive operation is determining the game outcome. Another example is an auction server, where the protocols are various auction techniques and the sensitive server operation is the exchange of goods. For example, the auction server can be extended with a new auction technique such as a time-limited public auction, e.g., the best price given in the next day.

### 3 Secure client-side API

This section describes *SecureAPI*, a secure client-side-only API. The idea behind SecureAPI, is to mimic the way that games are played in real-life, i.e., the players themselves assume the role of the server. If there is a disagreement among the players, then a randomly selected jury is appointed to determine if there is a hacker among the players.

Therefore, to better understand SecureAPI, we need to first understand the role of the server. Consider, for example, the role of a server in a game of poker:

1. Hold secret state (such as the shuffled cards, the hand of each player, etc).
2. Make sure each player makes his move on time.
3. Verify that all players follow the game rules.
4. Determine the game outcome.

Now consider a *real-life* poker game. Note that there is no central server; instead the players assume the role of the server, i.e., each player verifies that the other players follow the game rules/protocol. Only if there is a disagreement among players, then a video footage is presented to an impartial jury that finds out who broke the rules.

SecureAPI follows a similar idea. The server stores the game state, which is a key-value dictionary. Some of the entries in the dictionary may be visible only to certain players. The players manipulate the game state (store, shuffle or reveal entries), and decide when the game ends. All the players actions (except storing) must be unanimous, or else the server convenes a *jury* that should pinpoint the hacker. The jury is a set of users that are playing in another game and are therefore impartial. Each juror automatically (without human intervention) traces the game in question (similarly to the video footage analogy in real-life) and pinpoints the player who did not follow the game protocol.

Phrased differently, both *jurors* and *players* are programs that run on the user's machine (e.g., in Come2Play's platform it is a flash program running in

the user's browser), and connect over the internet with the *server*. The *player* program interacts with a human user using a graphical user interface (GUI), e.g., the user decides on his move in the chess game by dragging a certain chess piece to a certain location. In contrast, the *juror* program is deterministic and does not require any user input or any GUI. Therefore, the user is not even aware that he is running a juror program, or even several juror programs. A juror have nothing to gain by changing the game outcome. On the contrary, a juror that is not in agreement with the other jurors may be punished by the platform to discourage hacking attempts.

SecureAPI only assumption is that the percentage of hackers in the system is small (must be less than 50%). This assumption should be valid in real-life systems due to the following reasons:

1. In a system with thousands of online casual players, most players use the site for fun, and they will find it immoral to change the game outcome. They might also be reluctant to install any hacking software.
2. The hackers must form a large-enough coalition in order to have a statistically significant chance of hacking a game, i.e., the hackers must act in unison. For example, suppose that the system use a set of 3 jurors, and there is a coalition of hackers that comprise %1 of the players. Therefore, the probability that all 3 randomly chosen jurors are hackers is only 1 out of a million ( $\frac{1}{100} * \frac{1}{100} * \frac{1}{100} = \frac{1}{1000000}$ ). By increasing the number of jurors, the hacking probability is decreased exponentially.

### 3.1 Game-state

The most important role of the server is to maintain the game state. The state is a dictionary mapping a key to a `ServerEntry` object:

```
class ServerEntry {
    var key:Object, value:Object, storedByUserId:int,
        visibleToUserIds:int[], changedTime:int
}
```

Fields `key` and `value` are arbitrary objects, and equality between two `key` objects is defined as deep object equality (recursively comparing sub-objects).

Field `storedByUserId` is set to the id of the user who last stored/updated this entry.

Field `visibleToUserIds` determines if the server entry is public or partially visible, i.e., only a subset of the players receive its `value`. Specifically, if `visibleToUserIds` is `null` then the entry is *public*, i.e., visible to all. Otherwise, the entry is *secret* and only a subset of the players will get its value; the other players will still get the server entry but its `value` will be `null`.

Finally, field `changedTime` is the server's time when the entry was created or updated. Some games are time-dependent, e.g., turn-based games can limit the time-per-turn by using this field.

Here are some examples of using state in various multiplayer games (see Section 4 for more examples). In Multiplayer Snake, the state will contain the positions of the snakes on the board. In Chess, the state includes the positions of the pieces, who has the turn, and whether the king has moved from its initial position. In Poker, the state includes the deck of cards (which is not visible to any player), and the hand of each player (which is visible only to that player).

The players may perform the following operations on the state:

**Store** a value in the state. This will either create a new server entry, update one, or delete one.

**Shuffle** a set of keys. This will shuffle the values in a set of server entries, and will set `visibleToUserIds` to an empty array (i.e., make those entries invisible to all players).

**Reveal** a set of keys. This will add some players to `visibleToUserIds`.

Storing is done *individually* by each player, whereas shuffling or revealing is done *unanimously* by all players. Both shuffling and revealing changes `storedByUserId` to `-1`, so the players can differentiate between entries stored individually or collectively by all players.

For every state change, the server sends the modified entries to all the players, taking into consideration `visibleToUserIds`. The players should examine these entries to verify that players follow the game protocol and only store legal state.

For example, in a game of poker one player first stores the deck of cards in the state, then all players shuffle the keys, and finally all players deal the cards (i.e., reveal certain keys to each player).

## 3.2 Independent Games

We partition multiplayer games into two categories:

**Independent games** can be played without 3<sup>rd</sup> party help. Most real-life games fall into this category, such as Backgammon, Battleship, Poker, Multiplayer Snake, etc.

In most real-life games there is no additional entity that calculates a secret state; instead, the secret state stems either from the players' decisions (e.g., in Battleship, the initial positioning of your battleships) or from random reordering/shuffling/permuting of real-life objects such as cards, notes, pieces, etc.

**Dependent games** rely on 3<sup>rd</sup> party calculations. For example, in Multiplayer Sudoku or Minesweeper, the initial board is calculated by a 3<sup>rd</sup> party.

SecureAPI for independent games is described next, and Section 3.3 extends it to support dependent games.

SecureAPI, presented in API 1, is a simplified version of Come2Play’s API, i.e., SecureAPI has only the bare minimum to enable writing a multiplayer game. Specifically, it only handles a single match, it does not handle loading a saved game, nor does it handle players disconnecting in the middle of the game.

API 1 is a set of messages that can be transmitted between the server and a player. Messages that start with `got` (called *callbacks*) are received by the game, and those with `do` (called *operations*) are sent by the game.

```
1: gotMyUserId(myUserId:int, playerIds:int[])
2: doStoreState(userEntries:UserEntry[])
3: doAllShuffleState(keys:Object[])
4: doAllRevealState(revealEntries:RevealEntry[])
5: gotStateChanged(serverEntries:ServerEntry[])
6: doAllFoundHacker(userId:int, errorDescription:String)
7: doAllEndMatch()
```

**API 1:** SecureAPI for Independent Games

The API uses the following auxiliary classes:

```
class UserEntry {
    var key:Object, value:Object, isSecret:Boolean;
}
class RevealEntry {
    var key:Object, revealToUserIds:int[];
}
```

We next explain the semantics of each message:

**gotMyUserId** tells the game what is his id (`userId`), and what are the player’s ids (`playerIds`). If your id is not in the player’s ids, then you are only viewing the game and cannot perform any operations. The order of `playerIds` can be used to set the order of turns in a turn-based game.

**doStoreState** changes the state. If `value` is null, then an entry is deleted from the state. Otherwise, an entry is either created or modified. Let `e:ServerEntry` be the entry for the given key. Then, `e.storedByUserId` is the `userId` that called `doStoreState`. And, `e.visibleToUserIds` is either null if `isSecret` is false, or an array with only `userId`.

**doAllShuffleState** shuffles the values in the server entries of the given keys, sets `visibleToUserIds` to an empty array, and sets `storedByUserId` to `-1`.

**doAllRevealState** adds `revealToUserIds` to `visibleToUserIds`, and sets `storedByUserId` to `-1`.

**gotStateChanged** tells the game that the state changed.

**doAllFoundHacker** accuse another player of being a hacker. The server will convene a jury to confirm or reject this claim.

**doAllEndMatch** ends the game (the full API has parameters that determine the final score and pot percentage of each player).

Come2Play's API extends API 1 with 3 more operations (that can be implemented using existing operations):

```
doAllSetTurn(userId:int, millisecondsInTurn:int)
doAllStoreState(userEntries:UserEntry[])
doAllRequestRandomState(key:Object, isSecret:Boolean)
```

**doAllSetTurn** starts a timer on the server, and when the timer ticks **userId** will lose the game. It can be implemented by the players themselves, by starting a timer, and when the timer ticks storing in the state that the player's turn has timed-out, and ending the match. Because each **ServerEntry** in the state has the field **changedTime**, a jury can verify that the turn indeed ended in a timeout.

**doAllStoreState** is identical to **doStoreState** excepts that it must be called by all players (unanimous operation). It can be implemented by one player doing the store, and other players verifying the store is indeed performed (by starting a timer) and that the stored value is correct.

**doAllRequestRandomState** stores in the given key a random integer. This operation is useful in games that require random values, such as Roulette or the dice in Backgammon. It can be implemented by each player storing a random secret value, and then all players should call **doAllRevealState** to expose the secret random values. The final random value is obtained by XOR-ing all the random values.

### 3.3 Dependent Games

Dependent games relies on 3<sup>rd</sup> party *calculators*. For example, in Multiplayer Sudoku or Minesweeper, the initial board is calculated by a 3<sup>rd</sup> party. Some games that are independent in real-life, may be turned into a dependent game to prevent cheating.

Consider, for example, the *independent* version of Battleships. At the beginning of the game, each player secretly stores the chosen positions of his battleships. During the game, each player guesses the positions of his opponent's battleships, and *trusts* the opponent's answers. When the game ends, the secret positions are revealed, and the opponent's answers are verified. A hacker may cheat in this game by *not* storing any battleships. His opponent may give up on the game (because he will not be able to find any battleships) and disconnect before the game ends, thus the hacker wins the game before reaching the final verification stage in which the secret positions are revealed and verified.<sup>4</sup>

---

<sup>4</sup>We could add a post-mortem stage when a player disconnects, in which a jury checks the legality of the (secret) match state. However, that would only further complicate the API; we believe it is more intuitive to use calculators instead.

In the *dependent* version of Battleships, all players reveal their secret positions to a 3<sup>rd</sup> party that verifies that all players stored correct secret values. In this dependent version, the previous hacking attempt would fail, because instead of trusting the opponent's answer, all players reveal the relevant key and the server returns the entry value (whether a battleship was hit or missed).

API 2 presents the API for independent games; it extends API 1 by adding the following messages:

```
1: doAllRequestStateCalculation(keys:Object[])
2: gotRequestStateCalculation(serverEntries:ServerEntry[])
3: doAllStoreStateCalculation(userEntries:UserEntry[])
```

**API 2:** SecureAPI extension for Dependent Games

`doAllRequestStateCalculation` starts a request for 3<sup>rd</sup> party calculation. The server will pick a random set of users (called *calculators*), and send to all calculators the message `gotRequestStateCalculation` with `serverEntries` that match the `keys` given previously (ignoring the entry's `visibleToUserIds`). All calculators should send `doAllStoreStateCalculation` with identical entries. The server will then update the match state, where the `userId` of a calculator is considered to be `-1`.

If there is disagreement among calculators, then the server decides according to the majority of calculators, and the minority is suspected to be hackers (or it is a bug of the game developer).

### 3.3.1 Secrecy Theft by Calculators

In Section 1, we defined a *secure API* as one in which hackers cannot unlawfully change the game outcome. However a game can also be compromised if you discover your opponent secret state, e.g., if you know your opponent secret Battleship's positions. Recall that a calculator is given a portion of the (secret) state. Consider a game where one of the players is a hacker, and one of the randomly chosen calculators is a hacker as well. Then the hacker-calculator can perform *secrecy theft* by sending the hacker-player a portion of the secret state.

Therefore the gaming platform should choose calculators wisely:

1. The number of calculator should be small, either 2 or 3. There should be at least two calculators to prevent a single hacker from disrupting games.
2. The calculators should be among the trusted members of the community, e.g., highly rated players that visit the website often.

## 4 Examples

In *real-time* games (such as shooter-games or a multiplayer snake game), all players update the state simultaneously. In contrast, in a *turn-based* game

(such as Chess, Pool, and Poker) the players take turns in making moves, and each player updates the state only during his turn.

We will use the following games as examples of how to use SecureAPI:

**TicTacToe** is a classic deterministic *turn-based* game.

**Multiplayer Snake** is a deterministic *real-time* game.

**Backgammon, Roulette** are games that use *randomness* where all the game-state is *public*.

**Battleship, Stratego, Diplomacy** are *deterministic* games where each player has a *secret* state. For example, in Battleship and Stratego, the players first decide on an initial positioning. In Diplomacy, each player secretly writes down his next move, and all moves are unfolded only after all players finished writing their moves.

**Clue, Poker, Monopoly, Dominoes, Scrabble** have a secret state that stems from random permutations, i.e., shuffling the cards or game pieces.

**Multiplayer Sudoku, Minesweeper** are dependent games that use  $3^{rd}$  party calculators to create the initial board.

**TicTacToe** This is the easiest game to demonstrate, and we now give its high-level description. First all players call `doAllSetTurn` with the first id in `playerIds`. The game handles two events:

1. Receiving `gotStateChanged`, where the server entry contains a row and column values. The game must check the validity of the move, i.e., if it is indeed the turn of `storedByUserId`, if the row and column values are legal, if that cell is not already occupied, etc. If there are any exceptions, the game calls `doAllFoundHacker`. Otherwise, the graphics should be updated accordingly, and either a new turn starts (`doAllSetTurn`), or the game is over (`doAllEndMatch`).
2. Receiving a GUI event when the human user pressed on a certain cell (row and column). If the move is legal, the game should call `doStoreState` with the row and column values.

**Multiplayer Snake** In this real-time game, the players control a long, thin creature, resembling a snake, which roams around on a bordered plane, trying to avoid hitting its own tail, other snakes, or the "walls" that surround the playing area. The game-state is entirely public, and it contains the positions of the snakes. Each opponent's snake has a phantom part which predicts where the snake will go, and when the real position arrives, the graphics is updated. Colliding with the phantom does not end the game (because the opponent's snake might have moved).

**Backgammon** In this turn-based game, the game-state is public and the dice roll is random. The dice roll is performed by calling `doAllRequestRandomState` (see Section 3.2).

**Battleship** See the game description in Section 3.3.

**Clue** The real-life game of Clue has 21 cards: six different characters, six possible murder weapons, and nine different rooms. At the beginning of the game, the characters, weapons, and rooms, are shuffled, and one card from each is selected and placed in a sealed envelope. Then the rest of the cards are shuffled and divided between the players. The objective of the game is to discover the envelope's content by querying the other players on the cards that they were dealt. When a player believes he knows the envelope's content, he says aloud his hypothesis and check it (without showing the envelope's content to the other players). If he was wrong, he loses the game but the other players continue playing.

In the online version, the players first publicly store all 21 cards in the state, keys [1–6] are the characters, keys [7–12] are the murder weapons, and keys [13–21] are the rooms. Keys [1,7,13] will represent the envelope's content. All players make 4 calls to `doAllShuffleState`: (i) keys [1–6], (ii) keys [7–12], (iii) keys [13–21], and (iv) keys [2–6,8–12,14–21]. Then all players call `doAllRevealState` to reveal keys [2–6,8–12,14–21] to certain players. E.g., if there are two players, then keys [2–6,8–11] are revealed to the first player and keys [12,14–21] to the second player. When a player announces the envelope's content, all players reveal keys [1,7,13] to that player, and if he claims victory then the keys are revealed to the everyone to verify the claim.

**Multiplayer Sudoku** Sudoku is an example of a dependent game. All players first call `doAllRequestRandomState` to store a secret random seed in the state. Then they call `doAllRequestStateCalculation` and expose the seed to the calculators. Each calculator uses the seed he got in `gotRequestStateCalculation`, and stores a Sudoku board by calling `doAllStoreStateCalculation`. (Recall the calculators' output must be identical.) Some of the entries are public and some are secret, i.e., the secret cells represent the hidden sudoku cells that the players should discover.

The players try to fill the hidden Sudoku cells correctly, as fast as possible because this is a real-time game. When a hidden cell is first filled, all players call `doAllRevealState` to make that cell public in order to determine if the player who first filled the cell is correct or not.

## 5 Server Algorithm for SecureAPI

This section describes the server algorithm, starting with a high level description. We assume that all communication between the server and the user uses

SSL over TCP, thus no messages are lost, and a hacker cannot spoof messages or hijack a session.

The server performs a `doAll` message only if all players sent it with identical parameters. Therefore, if player  $p$  sent a `doAll` message, followed by other messages, then those other messages must be queued to be processed later (after all other players have sent the above `doAll` message).

Formally, the server maintains, for each player  $p$ , a queue of incoming messages  $q_p$ . Each queue entry contains the message and the time of its arrival. The server performs a *merge algorithm* over all queues using the time of arrival. Specifically, let  $M$  be the set of candidate messages, i.e., the messages from the top of each queue. If  $M$  contains a `doStoreState` message, then the server processes the one which has the minimal time of arrival. Otherwise, if  $M$  contains `doAll` messages from all players, then the server checks that all messages are identical (or it convenes a jury), and then processes the message. Otherwise, the server waits for the next message to arrive.

The server must guarantee timely progress to make sure that a hacker cannot cause stagnation by not sending a `doAll` message. In other words, if  $M$  contains a very old message then a jury is convened.

Note that the server algorithm immediately broadcasts a `doStoreState` message, unless that player previously sent a `doAll` message. This property is critical to real-time games that require fast response times.

Section 5.1 discusses important extensions to the API that are present in Come2Play's API. Section 5.2 discusses the process of convening a jury. Finally, Section 5.3 gives examples of hacking attempts and how the jury system catches them.

## 5.1 Extensions to The API and Server Algorithm

**Disconnecting Players** Come2Play's API supports disconnecting players. Consider a player  $p$  that disconnected in the middle of the game. Then the server sends a new message `gotMatchEnded(p)`, and it now treats  $p$  as a viewer, i.e., remove its queue  $q_p$  and not wait for his `doAll` operations.

**Loading Games and Transactions** Come2Play's API supports loading saved games and viewers that join the game in the middle by adding two new messages:

```
gotMatchStarted(playerIds:int [], state:ServerEntry[])
doStateChangeVerified()
```

Parameter `state` will either contain the saved game state (when loading a saved game) or the current game state (when a viewer joins the game). Message `doStateChangeVerified` is automatically sent after every `gotStateChanged` for two reasons: (i) to *verify* that the state change was legal (according to the rules of the game), and (ii) to group messages into *transactions*. The *verified-state* is used when convening a jury (see Section 5.2).

*Transactions* are used to perform state operations atomically. For example, in the beginning of a game of poker the players perform several state operations:

write, shuffle and deal the cards. These operations must be done atomically, or else the game must handle special cases when loading a saved game, e.g., a saved game that only wrote the cards but still did not shuffle them.

The API enforces that `doAll` messages may only be sent in a transaction as a reply to a server message. Phrased differently, only `doStoreState` may be sent as a result of a GUI event. For example, in TicTacToe, when a player makes his move, the game only sends `doStoreState`, and only when the game gets `gotStateChanged` it sends a `doAll` message (either `doAllSetTurn` or `doAllEndMatch`).

## 5.2 Convening a Jury

When the players disagree, the server convenes a jury to pinpoint the hacker. The disagreement is between at least two players, denoted `p` and `q`.

A jury is convened in three cases:

1. Player `p` sent `doAllFoundHacker(q)`.
2. Player `p` sent a `doAll` message that is not equal to another `doAll` message that was previously sent by `q`.
3. Player `p` sent a `doAll` message that timed-out, i.e., some player `q` did not send that message.

A jury is a collection of users (called jurors) involved in another game that automatically (without human intervention) determine whether player `p` or `q` is the hacker.

Each juror is treated as if he were player `p`, i.e., a juror will see entries that are visible to `p`. The server sends each juror the *verified-state* followed by all the messages in the *unverified-queue*. If the jury acts identically to `p`, then `q` is the hacker. Otherwise `p` is the hacker.

The *unverified-queue* contains all the `gotStateChanged` messages that have not been verified yet by all players. After all players call `doStateChangeVerified` then a message is dequeued and the *verified-state* is updated. The server maintains another timer that guarantees progress by checking that the *unverified-queue* does not contain a very old message.

If the jury itself is not unanimous, then we either have a hacker in the jury or a bug in the game. The server can then pick another jury, follow the majority decision, or cancel the game and inform the game developer.

The number of jurors should be a random value between 2 and the number of possible jurors. The distribution of this random value should be exponentially decreasing, i.e., 50% probability of having 2 jurors, 25% for 3 jurors, 12.5% for 4 jurors, etc. The algorithm must never set a top bound for the number of jurors because if the hackers know the top bound then they can (with a tiny probability) interrupt a game with zero-risk: suppose the maximal number of jurors is  $X$ . Then all hackers can communicate with a hacker's central server, and if there is a game where all  $X$  jurors are hackers (tiny probability, but it

can happen) then they can interrupt the game with zero-risk of being detected. The hackers, of course, cannot choose which game to interrupt because the jury is randomly selected.

### 5.3 Hacking Examples and Using a Jury

Suppose one of the players is a hacker. Then he could harm the game in several possible ways:

1. The hacker sent `doAllFoundHacker`. The server will convene a jury, and that jury will not send `doAllFoundHacker`, thus pinpointing the hacker.
2. The hacker sent an illegal `doStoreState`. Then the other players should eventually<sup>5</sup> call `doAllFoundHacker`, and the server will convene a jury. The jury will also call `doAllFoundHacker`, thus pinpointing the hacker.
3. The hacker sent an illegal `doAll` message. That message will either clash with another `doAll` message or it will timeout (because other players will not send that message). In both cases, a jury will not send that `doAll` message, thus pinpointing the hacker.
4. The hacker did *not* send a `doStoreState`. In a turn-based game, the hacker's turn will end and the hacker will lose. In a real-time game, the opponents can proceed without waiting for the hacker.
5. The hacker did *not* send a `doAll` message. Then the server will get a timeout and convene a jury. The jury will send that message, thus pinpointing the hacker.

## 6 Conclusions

This paper presented SecureAPI, which is the first secure client-side only API. SecureAPI mimics real-life games by moving the traditional server-side logic to the client-side, by making the players verify each other. There is no (server-side) overhead for using SecureAPI if none of the players is a hacker. When there is a hacker, then a jury is convened to automatically pinpoint the hacker.

SecureAPI was implemented by Come2Play, that developed an open-source emulator for the API. Several games have already been developed (multiplayer TicTacToe, Dominoes, MinesSweeper, and Snake), and advertising revenue is shared with the game developers.

---

<sup>5</sup>If the hacker sent illegal *secret* state, then the hacking will be discovered only when the secret is revealed.