On Security Against Time Traveling Adversaries

Lúcás Críostóir Meier
lucas@cronokirby.com

September 3, 2022

Abstract

In this work, we investigate the notion of time travel, formally defining a model for adversaries equipped with a time machine, and the subsequent consequences on common cryptographic schemes.

1 Introduction

[Mau09]

2 Defining Abstract Games

In this section, we develop a framework to abstract over essentially all security games used to define the standalone security of cryptographic schemes.

We need such an abstraction in order to explore and compare different models of time travel. By having an abstract game, we can more easily define what it means to augment an adversary with the ability to travel through time, and we can more easily compare the differences between models of time travel across *all* games, rather than for just a particular cryptographic scheme.

2.1 State-Separable Proofs

But first, we need a basic notion of standalone security. For this, we lean on the framework of *state-separable proofs* cite.

In this framework, you start with *packages*. A package P is defined by its code. This codes describes how to initialize the state of the package, and what functions the package exports. These exported functions are denoted by the set $\text{out}(P)$. Each of these functions can accept input, produce output, and read and write to the internal state of the package. Packages may also have a set of imported functions, denoted by $\text{in}(P)$.

These imported functions are just "placeholders", with no semantics. For them to have meaning, the package needs to be *linked* with another package. If A and

B are packages such that $\text{in}(A) \subseteq \text{out}(B)$, then we can define the composition package $A \circ B$. The exports are that of A, with $\text{out}(A \circ B) = \text{out}(A)$, and the imports that of B, with $\text{in}(A \circ B) = \text{in}(B)$. This package is implemented by merging the states of A and B, and replacing calls to the functions in $\text{in}(A)$ with the functionality defined in B.

A *game* G is a package with $\text{in}(G) = \emptyset$.

An *adversary* $\mathcal{A}$ is a package with $\text{out}(\mathcal{A}) = \{\texttt{guess}\}$. The function `guess` takes no input, and returns a single bit $\hat{b}$. This bit represents the adversary's guess as to which of two games it's playing. Furthermore, if the function `guess` has a time complexity polynomial in a security parameter λ, we say that the adversary is *efficient*. Most commonly, we assume that all adversaries are efficient, unless we explicitly mark an adversary as *unbounded*.

By linking an adversary with a game, we get a package $\mathcal{A} \circ G$ with no imports, and a single export `guess`. This allows us to define the advantage of an adversary $\mathcal{A}$ in distinguishing two games G_0, G_1, via the formula:

$$\epsilon(\mathcal{A} \circ G_b) := |P[1 \leftarrow \texttt{guess}() \mid b = 0] - P[1 \leftarrow \texttt{guess}() \mid b = 1]|$$

Given we a pair of games G_0, G_1, we say that they are:

- *equal*, denoted by $G_0 = G_1$, when $\epsilon(\mathcal{A} \circ G_b) = 0$ for any adversary, even unbounded.
- *indistinguishable*, denoted by $G_0 \approx G_1$, when $\epsilon(\mathcal{A} \circ G_b)$ is a negligeable function of λ, for any *efficient* adversary (in λ).

The security of a cryptographic scheme is defined by a pair of games G_b. We say that the scheme is *secure* if $G_0 \approx G_1$.

For reductions, given game pairs G_b and $H_b^1, \ldots, H_b^N$, and a function p, we write:

$$G_b \leq p(H_b^1, \ldots, H_b^n)$$

If for any efficient adversary $\mathcal{A}$ against G_b, there exists efficient adversaries $\mathcal{B}_1, \ldots, \mathcal{B}_n$ such that:

$$\epsilon(\mathcal{A} \circ G_b) \leq p(\epsilon(\mathcal{B}_1 \circ H_b^1), \ldots, \epsilon(\mathcal{B}_n \circ H_b^n))$$

2.2 Abstract Games

In the formalism of state-separable proofs, each game can have a different interface, and maintain a different kind of state. This is very useful, since it allows

us to capture various cryptographic schemes and notions of security. However, in order to easily model the impacts of time travel on various games, we would rather work with a *single* interface, capable of capturing the behavior of various games and their notions of security.

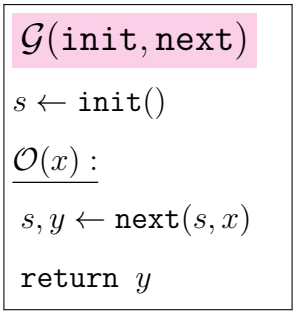
The key observation here is that the state of a pair of game is modified in only two places:

1. When the state is initialized.
2. When an exported function is called.

We can also collapse all of the exported functions into a single function, by including additional information in the input. For example, the input can include which sub-function is being called, along with the arguments to that sub-function.

The data we need to describe a game thus consist of a set of states Σ, an initialization function $\texttt{init} : () \xrightarrow{R} \Sigma$, as well as input and output types X and Y, along with a transition function $\texttt{next} : X \times \Sigma \to Y \times \Sigma$.

Together, these data define the following game:



Game 1: $\mathcal{G}(\texttt{init}, \texttt{next})$

Intuitively, the game uses `init` to randomly initialize the state, and then each subsequent oracle call triggers some kind of randomized calculation which modifies the state, and produces an output.

We can also implicitly parameterize the types and functions with a bit b, allowing us to define the game pair $\mathcal{G}_b(\texttt{init}, \texttt{next})$, which is shorthand for $\mathcal{G}(\texttt{init}_b, \texttt{next}_b)$.

This abstract game is simple, but still expressive enough to capture any kind of game expressable in the state-separable formalism.

3 Models of Time Travel

In this section we investigate various models of time travel, and compare them with each other, showing that they form a hierarchy of increasingly strong capabilities.

The notion of time travel we explore is an intuitive one, inspired by science fiction. The adversary is equipped with a time machine, which allows them to travel forwards and backwards in time. However, the adversary must still be *efficient*. From their point of view, they only perform a number of operations polynomial in the security parameter λ, including time travel hops.

Some other models of time travel, like closed timelike curves, would allow, in essence, for computation with unbounded time (but bounded space) by an adversary. This is a much more powerful capability than we consider in this work, and unbounded computation breaks essentially all cryptography beyond information-theoretic schemes.

We also assume that time is *discrete*. Each interaction the adversary has with a game advances time forward by one step, and time hops can only be made between these discrete points in time. One potentially stronger capability would be to allow an adversary to "partially" undo the effects of an interaction, by rewinding an interaction before its completion. The reason we disallow this is because we assume the adversary has no other channels to learn about the state of the game beyond the information it gets from querying its exported functions. An adversary thus has no way of knowing where they need to time hop in order to partially undo an interaction, so we can make the simplifying assumption that all interactions are *atomic*, and time is discrete.

3.1 Rewinding Models

The first model of time travel we consider is that of *rewinding*, in which the adversary is allowed to travel backwards in time.

3.1.1 Single Rewinds

We start by giving the adversary the ability to travel backwards by exactly one time step.

We model this as a *transformation* between games. Given an abstract game $\mathcal{G}_b$, we define the game Rewind-1$(\mathcal{G}_b)$ as follows:

Rewind-1($\mathcal{G}_b$)	
$s_0 \leftarrow \texttt{init}_b()$	
$i \leftarrow 0$	
$\underline{\mathcal{O}(x):}$	$\underline{\texttt{Rewind}():}$
$s_{i+1}, y \leftarrow \texttt{next}_b(s_i, x)$	$\texttt{assert } i > 0$
$\texttt{return } y$	$i \leftarrow i - 1$

Game 2: Rewind-1($\mathcal{G}_b$)

The interface is the same as that of $\mathcal{G}_b$, except that we now have an additional exported function: `Rewind`. Apart from this function, the behavior of the game is the same. Each interaction with $\mathcal{O}$ advances the state. The difference is only in the internal implementation. Instead of a single state s, we now have a sequence of states $s_0, s_1, \ldots$, as well as a position in this sequence, i.

The `Rewind` function is the additional capability here, and allows the adversary to move backwards by one step in time. This essentially models a very limited time machine, only allowing a small backwards movement in time.

Our first question is: does this limited model of time travel help the adversary? In other words, is an adversary with this capability more powerful than adversary without it? One way of capturing this notion of power would be to demonstrate a game $\mathcal{E}_b$ which is *secure*, but where Rewind-1($\mathcal{E}_b$) is broken. In fact, we can do this:

Claim 3.1. There exists a game $\mathcal{E}_b$ and adversary $\mathcal{A}$ such that $\mathcal{E}_b$ is secure, yet $\epsilon(\mathcal{A} \circ \text{Rewind-1}(\mathcal{E}_b)) = 1$.

Consider the following game:

$\mathcal{E}_b$	
$k_0, k_1 \xleftarrow{R} \{0,1\}^\lambda$	
$\text{queried} \leftarrow 0$	
$\underline{\texttt{Query}(\sigma):}$	$\underline{\texttt{Guess}(\hat{k}_0, \hat{k}_1):}$
$\texttt{assert } \text{queried} = 0$	$\texttt{assert } \hat{k}_0 = k_0 \wedge \hat{k}_1 = k_1$
$\text{queried} \leftarrow 1$	$\texttt{return } b$
$\texttt{return } k_\sigma$	

In this game, we have two random keys k_0, k_1. The game lets the adversary choose to learn one of the keys, but not the other. If the adversary manages to guess both of the keys, then they'll be able to learn the value of b.

Now, because k_σ has λ bits, an adversary won't be able to randomly guess its value. This means that if the adversary only knows one of the keys, they won't be able to pass the assertion except with negligeable probability. This means that $\mathcal{E}_b$ is secure.

On the other hand, Rewind-1$(\mathcal{E}_b)$ is already broken. The following strategy will always succeed:

$$
\begin{aligned}
&k_0 \leftarrow \texttt{Query}(0)\\
&\texttt{Rewind}()\\
&k_1 \leftarrow \texttt{Query}(1)\\
&b \leftarrow \texttt{Guess}(k_0, k_1)\\
&\texttt{return } b
\end{aligned}
$$

Even though the adversary prevents us from querying more than once, a single rewinding step is enough to undo our query, and thus learn the other key.

■

3.1.2 Multiple Rewinds

Next, we consider the ability to travel backwards by multiple steps at once. Like before, we model this with another transformation: Rewind-Many.

Rewind-Many$(\mathcal{G}_b)$	
$s_0 \leftarrow \texttt{init}_b()$	
$i \leftarrow 0$	
$\underline{\mathcal{O}(x):}$	$\underline{\texttt{Rewind}(j):}$
$s_{i+1}, y \leftarrow \texttt{next}_b(s_i, x)$	$\texttt{assert } i >= j$
$\texttt{return } y$	$i \leftarrow i - j$

Game 3: Rewind-Many$(\mathcal{G}_b)$

The only difference with Rewind-1 is that now the adversary can specify a hop distance j, and move backwards by j steps, rather than by just a single step.

A natural question arises: is being able to jump backwards multiple steps at a time more powerful?

No.

Claim 3.2. Rewind-Many is as strong as Rewind-1. In particular, for any abstract game $\mathcal{G}_b$, we have Rewind-Many$(\mathcal{G}_b) \leq$ Rewind-1$(\mathcal{G}_b)$.

The reduction works by emulating a large jump with many tiny jumps.

We define a wrapper Γ:

Γ

$\underline{\mathcal{O}(x):}$
`return super.`$\mathcal{O}(x)$

$\underline{\texttt{Rewind}(j):}$
`assert` $i >= j$
`super.Rewind()` j times

It then holds that:

$$\text{Rewind-Many}(\mathcal{G}_b) = \Gamma \circ \text{Rewind-1}(\mathcal{G}_b)$$

The only subtlety is that we need to guarantee that this emulation is efficient, i.e. polynomial in λ. Because the adversary for $\mathcal{A}$ against Rewind-Many$(\mathcal{G}_b)$ is efficient, we know that they make a number of queries to $\mathcal{O}$ polynomial in λ. This means that the largest i they reach is also bounded, and thus so will the largest j they query. This means that the number of iterations we do in the emulation is also bounded by a polynomial in λ, so the reduction is efficient.

■

3.2 Forking Models

So far, we've considered a simple model of time travel in which the adversary observes a linear sequence of states, but they're allowed to rewind time, undoing the most recent states.

figure?

One shortcoming of this model is that the adversary has no ability to return to previously seen states. For example, after reaching a state s, an adversary can move backwards in time, but then loses the ability to move back to the state s.

While they can travel backwards in time, they can't travel *forwards* at will.

In this section, we augment the adversary with the ability to travel both forwards and backwards and time. To do so, we consider a model in which the adversary is allowed to *fork* the timeline, and then travel between these parallel timelines. Instead of having a linear sequence of states, we now have a tree:

figure?

To model this technically, we introduce the notion of *savepoints*. By creating a savepoint at particular point in time, an adversary is able to return to the state of the game at that point in time. Each savepoint is thus a junction point in the tree. By returning back to a savepoint, the adversary creates a new branch at that junction:

figure

3.2.1 A Stack of Savepoints

In the first model we consider, an adversary is free to create savepoints anywhere, but can only jump to the most recently created savepoint. We denote this capability by Fork-Stack:

Fork-Stack($\mathcal{G}_b$)

$s \leftarrow \texttt{init}_b()$

stack $\leftarrow \varepsilon$

$\underline{\mathcal{O}(x)}$:

$s, y \leftarrow \texttt{next}_b(s, x)$

`return` y

`Fork()` :

stack.push(s)

`Load()` :

`assert` ¬stack.empty

$s \leftarrow$ stack.pop()

Game 4: Fork-Stack($\mathcal{G}_b$)

We maintain a stack of savepoints, which are just snapshots of the state of the game, but we can only reload the most recent savepoint, consuming it. The adversary also needs to proactively create savepoints if they want to be able to rewind time.

How does Fork-Stack compare with Rewind-Many?

It turns out that they're equivalent.

Claim 3.3. For all abstract games $\mathcal{G}_b$, we have both Fork-Stack($\mathcal{G}_b$) $\leq$ Rewind-Many($\mathcal{G}_b$) and Rewind-Many($\mathcal{G}_b$) $\leq$ Fork-Stack($\mathcal{G}_b$).

Proof Idea:
Because we can only load the most recent savepoint, we can emulate these loads using rewinding.

In the other direction, we need to emulate rewinding with forking. One tricky aspect is that an adversary can rewind to any point without having to create a savepoint there in advance. In particular, they can choose how they rewind based on the results of interacting with the game. To accomodate this freedom, we can simply always make a savepoint, allowing us to rewind by loading multiples times in a row.

Proof:
First, we show that Fork-Stack$(\mathcal{G}_b) \leq$ Rewind-Many$(\mathcal{G}_b)$.

We define a wrapper Γ:

Γ

$i \leftarrow 0$

stack $\leftarrow \varepsilon$

$\underline{\mathcal{O}(x):}$

$i \leftarrow i + 1$

`return super.`$\mathcal{O}(x)$

$\underline{\texttt{Fork}():}$

stack.push(i)

$\underline{\texttt{Load}():}$

`assert` ¬stack.empty

$\hat{i} \leftarrow$ stack.pop()

`super.Rewind`$(i - \hat{i})$

$i \leftarrow \hat{i}$

This wrapper satisfies:

$$\text{Fork-Stack}(\mathcal{G}_b) = \Gamma \circ \text{Rewind-Many}(\mathcal{G}_b)$$

Basically, instead of keeping a stack of states, we can keep a stack of indices, and the rewinding is enough to load previous states, because we can only ever load the most recent state on the stack.

Next, we show that Rewind-Many$(\mathcal{G}_b) \leq$ Rewind-Many$(\mathcal{G}_b)$.

In Claim 3.2, we showed that Rewind-Many$(\mathcal{G}_b) \leq$ Rewind-1$(\mathcal{G}_b)$, so it suffices to prove that Rewind-1$(\mathcal{G}_b) \leq$ Fork-Stack$(\mathcal{G}_b)$.

We define a wrapper Γ, which works by always creating a savepoint, and then using those to implement rewinding.

Γ

$\underline{\mathcal{O}(x):}$

`super.Fork()`

`return super.`$\mathcal{O}(x)$

$\underline{\texttt{Rewind}():}$

`super.Load()`

We have:

$$\text{Rewind-1}(\mathcal{G}_b) = \Gamma \circ \text{Fork-Stack}(\mathcal{G}_b)$$

One subtlety is that in `Rewind`, we don't perform any assertions, whereas we'd usually check that $i > 0$. This isn't necessary because `super.`$Load$ will check that the stack isn't empty, which performs this duty.

■

3.2.2 Arbitrary Savepoints

In the Fork-Stack model, the adversary is limited to only load the most recently created savepoint. As we proved in Claim 3.3, this model gives no advantage over just being able to move backwards in time.

In order to capture the ability to move both backwards and forwards at will, we can remove this restriction on which savepoints can be loaded. We now maintain a list of savepoints, and these savepoints can loaded in any order and multiple times, at will, without any restrictions.

More formally, we capture this notion with the Fork transformation:

$\text{Fork}(\mathcal{G}_b)$

$s_0 \leftarrow \texttt{init}_b()$

$i, j \leftarrow 0$

$\underline{\mathcal{O}(x):}$

$s_i, y \leftarrow \texttt{next}_b(s_i, x)$

`return` y

$\underline{\texttt{Fork}():}$

$j \leftarrow j + 1$

$s_j \leftarrow s_i$

$\underline{\texttt{Load}(i'):}$

`assert` $i' \leq j$

$i \leftarrow i'$

Game 5: Fork($\mathcal{G}_b$)

The essence is that the game now maintains multiple states $s_0, s_1, \ldots$ in parallel. At any point, the adversary is free to switch which state is currently being used, or to create a parallel state from the current one. This captures the intuitive notion of traveling at will between parallel timelines.

It turns out that this model of time travel is strictly stronger than the others we've seen so far.

Claim 3.4. Fork is strictly stronger than Fork-Stack, assuming the existence of secure pseudo-random functions.

In particular, given a secure PRF F, there exists a game $\mathcal{E}_b$ and adversary $\mathcal{A}$ such that Rewind-1$(\mathcal{E}_b)$ is secure, yet $\epsilon(\mathcal{A} \circ \text{Fork}(\mathcal{E}_b)) = 1$.

Consider the following game:

$\mathcal{E}_b$

	$\underline{\texttt{Query}(\hat{x}):}$
$k \xleftarrow{R} \mathcal{K}$	$\texttt{assert}$ queried $= 0$
$x \leftarrow \perp$	queried $\leftarrow 1$
queried $\leftarrow 0$	$\texttt{return}$ $F(k, \hat{x})$
$\underline{\texttt{Win}(y):}$	$\underline{\texttt{Input}():}$
$\texttt{assert}$ $x \neq \perp$	$\texttt{assert}$ queried $= 0$
$\texttt{assert}$ $F(k, x) = y$	queried $\leftarrow 1$
$\texttt{return}$ b	$x \xleftarrow{R} \mathcal{X}$
	$\texttt{return}$ x

The idea is that we have a pseudo-random function, seeded with a random key k. The adversary can either query the function on an input of their choice, or attempt to win the game, by receiving a challenge input x, and then responding with the evaluation of the PRF F on that input. Crucially, they're allowed to perform a query, or to prepare the challenge input, but not both.

This game is insecure against forking, as demonstrated by the following strategy for Fork$(\mathcal{E}_b)$:

$x \leftarrow \texttt{Input}()$
$\texttt{Fork}()$
$\texttt{Load}(0)$
$y \leftarrow \texttt{Query}(x)$
$\texttt{Load}(1)$
$b \leftarrow \texttt{Win}(y)$
$\texttt{return } b$

On the other hand, the game Rewind-1$(\mathcal{E}_b)$ remains secure, provided that $|\mathcal{X}|^{-1}$ is negligeable in λ. While the adversary can use rewinding to query multiple times, they won't know which input x they need to query. Except with negligeable probability, each new call to `Input` will yield a different value of x. Because the adversary cannot predict the value of x, nor can they learn the output of F after they know x, since F is a secure PRF, they cannot win the game.

■

3.3 Summary

4 On Depth and Position Restrictions

5 Effects of Time Travel on Common Schemes

5.1 Stateless Schemes Remain Secure

5.2 On Encryption

5.3 On Signatures

6 Further Work

7 Conclusion

References

[Mau09] Ueli Maurer. Unifying Zero-Knowledge Proofs of Knowledge. In *AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 272–286. Springer, Berlin, Heidelberg, 2009.