On Security Against Time Traveling Adversaries

Lúcás Críostóir Meier
lucas@cronokirby.com

September 4, 2022

Abstract

If you had a time machine, what cryptography would you be able to break?

In this work, we investigate the notion of time travel, formally defining a model for adversaries equipped with a time machine, and exploring the consequences for cryptography. We find that being able to rewind time breaks some cryptographic schemes, and being able to freely move both forwards and backwards in time breaks even more schemes.

We look at the impacts of time travel on encryption and signatures in particular, finding that the IND-CCA and EUF-CMA security games are broken, while IND-CPA and UUF-CMA remain secure.

1 Introduction

Classical cryptography tries to build schemes which are secure against adversaries equipped with a computer, so long as they're bounded in the amount of computation they can do. Post-quantum cryptography additionally provides adversaries with a *quantum* computer. While this is enough to break many existing schemes, it is nonetheless possible to design cryptography which can resist quantum attacks.

If time machines were possible, could they help an adversary break certain cryptographic schemes? Can we design cryptography which is secure even against time traveling adversaries? We investigate these questions in this work.

We consider the "intuitive" notion of time travel that one might encounter in various works of science fiction. An adversary has a time machine which allows them to jump through time, rewinding time backwards, or skipping time forwards. Crucially, the adversary is still bounded in the computation they can do, including the amount of time travel. From their perspective, they still run in polynomial time.

This capability allows the adversary to break certain schemes, because they're able to get around various checks made in security games. For example, in the EUF-CMA game, the adversary has access to an oracle allowing them to sign any message of their choice. The adversary wins by producing a forgery for a new message, that hasn't been signed by the oracle. Time travel enables an adversary to trivially break this game by querying the oracle, then traveling backwards in time, to a point where no messages have yet been signed, and then presenting their signature as a valid forgery. Intuitively, this is akin to convincing someone that they've signed a message by presenting them with a signature that they will create in the future.

Beyond investigating concrete breakages of certain schemes, we also present a fine grained analysis of different models of time travel. We consider several variations of time travel, and show how they form a hierarchy, with some models strictly more powerful than others. In particular, the ability to travel backwards is more powerful than no time travel at all, and being able to travel both forwards and backwards is even more powerful.

We also consider restrictions on which points in time the adversary can travel to, and how deep they can travel in forked timelines, showing that these restrictions are strictly weaker.

1.1 Relevance of Time Travel

Given that time machines are mainly a subject of science fiction, the notion of time travel might seem irrelevant to cryptography in practice. We think that this might not be the case.

One notion similar to that of time travel is "resettable encryption" [Yil09]. In this work, they consider a cryptographic system deployed on a virtual machine. Because the adversary has the ability to reset the state of the virtual machine, the system needs to satisfy a higher level of security, in order to withstand attacks using this resetting functionality. From the perspective of the system, the adversary effectively has the ability to time travel, by rewinding the state backwards in time. Our models of time travel may thus be relevant to certain deployments of cryptography on virtual machines, when the adversary has the ability to rewind the state of that machine.

Other attacks against practical systems have been conducted by tricking devices into rewinding their system clocks backwards in time. For example, an attack by Iovino et al. [IVV20] used this clock corruption in order to attack digital contact tracing systems. From the perspective of the system under attack, clock rewinding adversaries are effectively traveling backwards in time. Thus, our exploration of time travel might be relevant to practical systems whose security depends on a clock.

Finally, in the realm of probabilistic proofs, notions of rewinding are relevant to security. For proofs of knowledge, the extractor is often given the ability to *rewind* the prover, and uses this ability to extract a secret value from the prover. From the perspective of the prover, the extractor is effectively traveling back in time. For interactive oracle proofs [BSCS16], the notion of *state-restoration soundness* is relevant when compiling a multi-round interactive protocol into a non-interactive proof. Indeed, the protocol must remain sound even if a prover has the ability to retry each round individually. We can model this by giving the prover access to a time machine which allows them to jump backwards in time. Thus, the crucial notion of state-restoration soundness coincides with notions of time travel we explore in this work.

1.2 Overview

In Section 2, we revisit the notion of standalone security, and define the notion of *abstract games*, which allow us to encapsulate security games under a common interface.

In Section 3 we then use this notion to define various models of time travel as transformations of abstract games. We compare these various models, showing that they form a hierarchy, with some models strictly more powerful than others.

In Section 4, we consider the effects of restrictions on the positions where an adversary can create forked timelines, and on the depth of each timeline, showing that these restrictions create strictly weaker capabilities.

In Section 5, we investigate the effect of time travel on common cryptographic schemes. We find that schemes whose security can be captured by a *stateless* game are unaffected by time travel, including IND-CPA security for encryption, and UUF-CMA security for signatures. On the other hand, both the IND-CCA and EUF-CMA games are broken, even in our weakest model of time travel.

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 just for 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* [BDLF$^+$18, Mei22].

The basic component of this framework is the *package*. A package P is defined by its code. This code describes how to initialize the state of the package, and what functions the package exports. This set of exported functions is denoted by $\text{out}(P)$. Each of these functions can accept input, produce output, and read and write to the internal state of the package. Packages 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 those of A, with $\text{out}(A \circ B) = \text{out}(A)$, and the imports are those 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 they're 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 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 negligible function of λ, for any *efficient* adversary (in λ).

The security of a cryptographic scheme is captured 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 all of these games.

The key observation here is that the state of a 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:

$$
\boxed{
\begin{array}{l}
\mathcal{G}(\texttt{init}, \texttt{next}) \\
s \leftarrow \texttt{init}() \\
\underline{\mathcal{O}(x):} \\
\quad s, y \leftarrow \texttt{next}(s, x) \\
\quad \texttt{return } y
\end{array}
}
$$

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 expressible 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 [Bru03]. 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 they get 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 models a very limited time machine which can only make small backwards movements 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 an 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.

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:

$\mathcal{E}_b$

$k_0, k_1 \xleftarrow{R} \{0,1\}^\lambda$

queried $\leftarrow 0$

$\underline{\texttt{Query}(\sigma):}$	$\underline{\texttt{Guess}(\hat{k}_0, \hat{k}_1):}$
$\texttt{assert}\ \text{queried} = 0$	$\texttt{assert}\ \hat{k}_0 = k_0 \land \hat{k}_1 = k_1$
$\text{queried} \leftarrow 1$	$\texttt{return}\ b$
$\texttt{return}\ k_\sigma$	

In this game, we have two random keys k_0, k_1. The game lets the adversary learn one key of their choice, but not the other. If the adversary manages to guess both of the keys, then they 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 negligible 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

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

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

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

$\texttt{assert } i >= j$

$\texttt{super.Rewind}()$ j times

It then holds that:

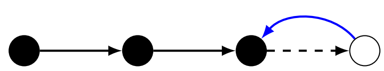
$$\text{Rewind-Many}(\mathcal{G}_b) = \Gamma \circ \text{Rewind-1}(\mathcal{G}_b)$$

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

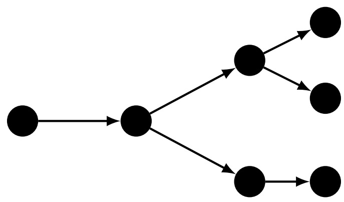
So far, we've considered a simple model of time travel in which the adversary observes a linear sequence of states, but they're allowed to rewind time, undoing the most recent states.



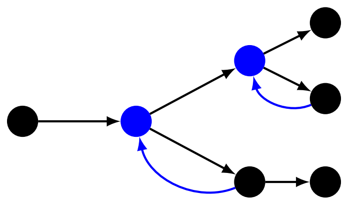
One shortcoming of this model is that the adversary has no ability to return to previously seen states. For example, after reaching a state s, an adversary can move backwards in time, but then loses the ability to move back to the state s.

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 in 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:



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



3.2.1 A Stack of Save Points

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

Fork-Stack$(\mathcal{G}_b)$	
$s \leftarrow \texttt{init}_b()$	$\underline{\texttt{Fork}() :}$
stack $\leftarrow \varepsilon$	stack.push(s)
$\underline{\mathcal{O}(x) :}$	$\underline{\texttt{Load}() :}$
$s, y \leftarrow \texttt{next}_b(s, x)$	$\texttt{assert}$ $\neg$stack.empty
$\texttt{return}$ y	$s \leftarrow$ stack.pop$()$

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

We maintain a stack of save points, which are just snapshots of the state of the game, but we can only reload the most recent save point, consuming it. The adversary also needs to proactively create save points 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 save point, 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 save point there in advance. In particular, they can choose how they rewind based on the results of interacting with the game. To accommodate this freedom, we can simply always make a save point, allowing us to rewind by loading multiple 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:

<table>
<tr><td colspan="2">Γ</td></tr>
<tr>
<td>

$i \leftarrow 0$

stack $\leftarrow \varepsilon$

<u>$\mathcal{O}(x)$:</u>

$i \leftarrow i + 1$

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

</td>
<td>

<u>`Fork`() :</u>

stack.push(i)

<u>`Load`() :</u>

`assert` $\neg$stack.empty

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

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

$i \leftarrow \hat{i}$

</td>
</tr>
</table>

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 save point, and then using those to implement rewinding.

Γ	
$\underline{\mathcal{O}(x):}$	$\underline{\texttt{Rewind}():}$
$\texttt{super.Fork}()$	$\texttt{super.Load}()$
$\texttt{return super.}\mathcal{O}(x)$	

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 Save Points

In the Fork-Stack model, the adversary is limited to only loading the most recently created save point. 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 the restriction on which save points can be loaded. We now maintain a list of save points, and these save points can loaded in any order and multiple times, at will, without any restrictions.

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

Fork($\mathcal{G}_b$)

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

$i, j \leftarrow 0$

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

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

$\texttt{return } y$

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

$j \leftarrow j + 1$

$s_j \leftarrow s_i$

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

$\texttt{assert } i' \leq j$

$i \leftarrow i'$

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:

$\mathcal{E}_b$

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

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($\mathcal{E}_b$):

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

On the other hand, the game Rewind-1($\mathcal{E}_b$) remains secure, provided that $|\mathcal{X}|^{-1}$ is negligible 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 negligible 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

To summarize our findings, we have the following hierarchy of models of time travel:

$$\text{No-Time-Travel} < \text{Rewind-1} = \text{Rewind-Many} = \text{Fork-Stack} < \text{Fork}$$

So, even a bit of time travel helps, but then the next jump in capability only comes with the ability to fork timelines and travel at will between them. In other words, being able to jump backwards helps, and being able to jump forwards too helps even more.

4 On Depth and Position Restrictions

In the models we've considered so far, there are limits on what time travel capabilities the adversary has, but not in how they can use them. The adversary can fork whenever they want, as many times as they want, and advance each forked timeline at will.

In this section, we model these kinds of restriction on time travel, and compare how they relate to each other.

4.1 Modeling Restrictions

The first kind of restriction is on the *position* where an adversary can fork. Without time travel, the sequence of states $s_0, s_1, \ldots$ is indexed by $\mathbb{N}$. A natural restriction is to only allow the adversary to fork on a subset $P \subseteq \mathbb{N}$ of these states. For example, the adversary may only be allowed to fork on the initial state s_0, but not on any other state.

The second kind of restriction is on the *depth* of the forks. In our model, the adversary is free to explore each fork to any depth. They can advance the state in each fork arbitrarily. With this restriction, we instead only allow the adversary to advance the state in a forked timeline d times.

More formally, given a set of positions $P \subseteq \mathbb{N}$, and a depth $d \in \mathbb{N} \cup \{\infty\}$, we can define the following transformation $\text{Fork}(P, d)$:

$\text{Fork}(P, d)(\mathcal{G}_b)$

$$
\begin{array}{ll}
s_0 \leftarrow \texttt{init}_b() & \\
p_0 \leftarrow 0 & \underline{\texttt{Fork}():} \\
c_0 \leftarrow \infty & \quad \texttt{assert } p_i \in P \\
i, j \leftarrow 0 & \quad j \leftarrow j + 1 \\
\underline{\mathcal{O}(x):} & \quad s_j, p_j, c_j \leftarrow s_i, p_i, d \\
\quad \texttt{assert } c_i > 0 & \\
\quad c_i \leftarrow c_i - 1 & \underline{\texttt{Load}(i'):} \\
\quad p_i \leftarrow p_i + 1 & \quad \texttt{assert } i' \leq j \\
\quad s_i, y \leftarrow \texttt{next}_b(s_i, x) & \quad i \leftarrow i' \\
\quad \texttt{return } y &
\end{array}
$$

Game 6: $\text{Fork}(\mathcal{G}_b)$

This game is like $\text{Fork}(\mathcal{G}_b)$, except with a few more restrictions.

First, we keep track of the position of each fork along the timeline, via p_i. This allows us to prevent forks unless the position p_i is contained in the set of allowed positions P.

In order to restrict the depth of each fork, each fork is associated with a counter c_i, which decrements each time the state advances. The main timeline has its counter c_0 set to ∞, to allow arbitrary progression.

4.2 Comparing Restrictions

For certain parameters, $\text{Fork}(P, d)$ is actually equivalent to not having access to time travel. If $P = \emptyset$, or $d = 0$, then forking becomes impossible. This means that $\text{Fork}(\emptyset, d) = \text{Fork}(\emptyset, d')$, and similarly for $d = 0$.

In fact, an equivalent condition is that P contains no elements in $\text{poly}(\lambda)$, i.e. bounded by a polynomial in the security parameter λ. For example, if $P = \{2^\lambda\}$, then the set isn't empty, but all of the indices contained therein are unreachable, making time travel impossible.

At the other extreme, $\text{Fork}(\mathbb{N}, \infty) = \text{Fork}$. If we can fork anywhere, and to any depth, we recover the general model of forking defined previously. In fact, it suffices that $d \notin \text{poly}(\lambda)$. If the depth is larger than any polynomial in λ, then it's impossible for the adversary to ever exhaust it, rendering it effectively infinite. For P, if for all $p \in \mathbb{N}/P$, we have $p \notin \text{poly}(\lambda)$, then P is effectively equivalent to $\mathbb{N}$, since the forbidden positions aren't reachable by an efficient adversary.

It's clear that as P and d grow larger, the adversary grows more powerful. In particular, for all abstract games $\mathcal{G}_b$, and parameters P, P', d, d', we have:

- If $P \subseteq P'$, then $\text{Fork}(P, d)(\mathcal{G}_b) \leq \text{Fork}(P', d)(\mathcal{G}_b)$.
- If $d \leq d'$, then $\text{Fork}(P, d)(\mathcal{G}_b) \leq \text{Fork}(P, d')(\mathcal{G}_b)$.

But, is it possible that certain parameter values are equivalent? If we increase the size of the parameters, is that a strictly stronger capability?

It is. Increasing P and d yields a strictly stronger adversary.

Claim 4.1. For all $d > 0$, if $P'/P \neq \emptyset$, then there exists a game $\mathcal{E}_b$ and adversary $\mathcal{A}$ such that $\text{Fork}(P, d)(\mathcal{E}_b)$ is secure, yet $\epsilon(\mathcal{A} \circ \text{Fork}(P', d)(\mathcal{E}_b)) = 1$.

The basic idea of the proof is that we engineer a game which requires the adversary to fork at a position in P'/P, which demonstrates the separation.

Given $p \in P'/P$, we can construct the following game:

$\mathcal{E}_b$

$k_0, k_1 \xleftarrow{R} \{0,1\}^\lambda$

$i \leftarrow -1$

$\underline{\texttt{Query}(\sigma):}$	$\underline{\texttt{Guess}(\hat{k}_0, \hat{k}_1):}$
$i \leftarrow i + 1$	$\texttt{assert}\ \hat{k}_0 = k_0 \land \hat{k}_1 = k_1$
$\texttt{if}\ i \neq p$	$\texttt{return}\ b$
$\quad\texttt{return}\ \perp$	
$\texttt{return}\ k_\sigma$	

In order to win the game, the adversary needs to learn both k_0 and k_1. Because of their size, this requires the adversary to make two queries, both at step p. This requires the adversary to be able to fork at step p, which they are unable to do in $\text{Fork}(P, d)(\mathcal{E}_b)$.

■

Claim 4.2. For all P with $\min(P) \in \text{poly}(\lambda)$[1], if $d' > d$, and $d' \in \text{poly}(\lambda)$, then

[1]This condition means that there exists an element in P bounded by a polynomial in λ, so that P isn't effectively empty.

there exists a game $\mathcal{E}_b$ and adversary $\mathcal{A}$ such that Fork$(P, d)(\mathcal{E}_b)$ is secure, yet $\epsilon(\mathcal{A} \circ \text{Fork}(P, d')(\mathcal{E}_b)) = 1$.

The idea is to make a game which requires forking, and then advancing the state a larger number of steps, which requires the ability to reach a depth of d'. We can do this by requiring the adversary to guess two keys k_0 and k_1. In order to enforce a certain depth, we require the adversary to first choose their index σ, and then wait a certain number of steps before learning k_σ.

Given $p \in P$, we define the following game:

$\mathcal{E}_b$	
$k_0, k_1 \xleftarrow{R} \{0,1\}^\lambda$	$\underline{\texttt{Wait}():}$
$c \leftarrow \infty$	$c \leftarrow c - 1$
queried $\leftarrow 0$	$\texttt{if}\ \ c = 0 \land \sigma \neq \bot$
$\underline{\texttt{Query}(\hat{\sigma}):}$	$\quad\texttt{return}\ \ k_\sigma$
$\texttt{assert}\ \ \text{queried} = 0$	$\underline{\texttt{Guess}(\hat{k}_0, \hat{k}_1):}$
queried $\leftarrow 1$	$\texttt{assert}\ \ \hat{k}_0 = k_0 \land \hat{k}_1 = k_1$
$c \leftarrow d' - 1$	$\texttt{return}\ \ b$
$\sigma \leftarrow \hat{\sigma}$	

In order to learn k_σ, the adversary first commits to their choice of σ in `Query`, and then they need to call `Wait` $d' - 1$ times before learning the result.

A winning strategy against Fork(P, d') would be:

$\texttt{Wait}()$ p times
$\texttt{Fork}()$
$\texttt{Query}(0)$
$k_0 \leftarrow \texttt{Wait}()$ d' times
$\texttt{Load}(1)$
$\texttt{Query}(1)$
$k_1 \leftarrow \texttt{Wait}()$ d' times
$\texttt{Load}(1)$
$b \leftarrow \texttt{Guess}(k_0, k_1)$
$\texttt{return}\ \ b$

Notably, this strategy requires a forking depth of at least d', in order to be able to make the queries to `Wait`. One subtlety is that we need to wait at the start of the game in order to advance the state p times, at which point we're allowed to fork, since $p \in P$.

On the other hand, since $d < d'$, Fork$(P, d)(\mathcal{E}_b)$ is secure. The adversary cannot learn both k_0 and k_1 via `Query` and `Wait`, since they lack the depth in their fork. Since both keys have length λ, the adversary cannot guess either of them with more than negligible probability either.

■

5 Effects of Time Travel on Common Schemes

In this section, we explore the impacts of time travel on various cryptographic schemes.

First, we prove a general result, namely that for *stateless* games, time travel provides no advantage.

Then, we show a few more concrete results, in particular that while IND-CCA encryption is broken against time travel, IND-CPA encryption remains secure. We also investigate signatures, showing that the EUF-CMA game is broken against time travel, but that UUF-CMA remains secure.

5.1 Stateless Schemes Remain Secure

So far, we've shown that two models of time travel, Rewind-1, and Fork provide strictly stronger capabilities. The separation in both cases relied on the adversary being able to "undo" certain checks made inside of the game.

However, if the state of the game remains static after initialization, then the adversary gains no advantage through time travel, because the state never changes, so time travel has no effect on this state.

More formally, given an abstract game $\mathcal{G}_b(\texttt{init}, \texttt{next})$, we say that the game is *stateless* if for all inputs and states s, x, it holds that:

$$(s', \cdot) \leftarrow \text{next}(s, x) \implies s' = s$$

In other words, no matter what initial state we have, and what input we pass to the game, the state will never change.

Claim 5.1. For any stateless game $\mathcal{G}_b$, time travel provides no advantage.

In particular, we have Fork$(\mathcal{G}_b) \leq \mathcal{G}_b$.

Since the state never changes, we can easily emulate the time jumps by doing nothing. More formally, if we write down the Fork$(\mathcal{G}_b)$ game explicitly, using the fact that the state doesn't change, we get the game Γ^0:

$$
\boxed{
\begin{array}{ll}
\Gamma^0 & \\
s_0 \leftarrow \texttt{init}_b() & \underline{\texttt{Fork}():} \\
i, j \leftarrow 0 & \quad j \leftarrow j + 1 \\
\underline{\mathcal{O}(x):} & \underline{\texttt{Load}(i'):} \\
\quad \cdot, y \leftarrow \texttt{next}_b(s_0, x) & \quad \texttt{assert } i' \leq j \\
\quad \texttt{return } y & \quad i \leftarrow i'
\end{array}
}
$$

In Γ^0, the `Fork` and `Load` functions have no impact on the rest of the game, allowing us to separate out $\mathcal{G}_b$, to get:

$$
\Gamma^0 = \boxed{
\begin{array}{ll}
\Gamma^1 & \\
 & \underline{\texttt{Fork}():} \\
i, j \leftarrow 0 & \quad j \leftarrow j + 1 \\
\underline{\mathcal{O}(x):} & \underline{\texttt{Load}(i'):} \\
\quad \texttt{return super}.\mathcal{O}(x) & \quad \texttt{assert } i' \leq j \\
 & \quad i \leftarrow i'
\end{array}
} \circ \mathcal{G}_b
$$

which ends our reduction.

■

The security of many schemes can be formulated with a stateless game, so Claim 5.1 is a very useful tool to quickly show security against time travel. We make use of this tool frequently in the following sections.

5.2 On Encryption

One very common notion of security for encryption schemes is that of IND-CCA security [NY90]. In the IND family of games, the adversary can present the challenger with a message of their choice, receiving either the encryption of that message, or a random message [2]. We call this ciphertext a *challenge*. The difference between the variants IND, IND-CPA, and IND-CCA lies in what additional oracles the adversary has access to.

[2]We use the "real or random" variant of IND, rather than the "left or right" variant, since the former is easier to use with state-separable proofs.

In IND-CPA, the adversary has access to an oracle which lets them receive encryptions of messages of their choice. In IND-CCA, the adversary additionally has access to an oracle allowing them to decrypt ciphertexts of their choice. Crucially, the adversary is *not* allowed to decrypt any challenge ciphertexts.

Both of these games are formally presented in the appendix, as Game 7 and Game 8, respectively.

This difference is what allows time travel to break the IND-CCA game. Because the game keeps track of which challenge ciphertexts have been produced, disallowing decryption queries to those ciphertexts, the adversary can use time travel to make the game "forget" which challenges have been produced, and then use the decryption oracle to win.

Claim 5.2. Given any encryption scheme $\mathcal{E} = (\mathcal{K}, \mathcal{M}, \mathcal{C}, E, D)$, and message $m \in \mathcal{M}$, there exists a an adversary $\mathcal{A}$, such that:

$$\epsilon(\mathcal{A} \circ \text{Rewind-1}(\text{IND-CCA}_b(\mathcal{E}))) = 1 - |\mathcal{M}(|m|)|^{-1}$$

where $|\mathcal{M}(|m|)|$ denotes the number of messages with the same length as m.

The idea is quite simple: the adversary first obtains a challenge ciphertext (any message works), and then makes the game forget that it produced this challenge, allowing the adversary to then query the decryption oracle on the challenge, thus learning information about the secret bit b.

More formally, consider the following strategy:

$$\begin{aligned}
&c \leftarrow \texttt{Challenge}(m)\\
&\texttt{Rewind}()\\
&\hat{m} \leftarrow \texttt{Dec}(c)\\
&\texttt{return } \hat{m} \neq m
\end{aligned}$$

The adversary will be allowed to query Dec with c, because the set of challenge ciphertexts is made empty by `Rewind`. Then, if $b = 0$, the adversary will return 0, since the ciphertext will be an encryption of m. Otherwise, if $b = 1$, the adversary will only return 0 if the random message happens to equal m, which happens with probability $|\mathcal{M}(|m|)|^{-1}$.

■

Crucially, our attack uses the fact that the IND-CCA game is stateful. IND-CPA, on the other hand, remains secure, because the game is inherently stateless.

Claim 5.3. For any encryption scheme $\mathcal{E}$, Fork(IND-CPA($\mathcal{E}$)) $\leq$ IND-CPA($\mathcal{E}$).

Looking at the definition in Game 7, it's clear that the associated abstract game is *stateless*. We can thus apply Claim 5.1 to obtain our result.

■

5.3 On Signatures

Next, we consider the security of signatures. One common family of security games for signatures is that of $*$UF-CMA [GMR88]. In this family, the adversary has access to an oracle which can sign messages, and they attempt to create a forged signature on a message. The difference between the games lies in which messages the adversary needs to forge a signature for.

In UUF-CMA, the game chooses a random message m, and the adversary must forge a signature for m. Naturally, the adversary is not allowed to use the signing oracle on m.

In EUF-CMA, the adversary can forge a signature for any message of their choice, so long as they didn't use that message with the signing oracle.

Both of these games are formally presented in the appendix, as Game 9 and Game 10, respectively.

In EUF-CMA, the game needs to keep track of which messages it has signed for the adversary. This bookkeeping is what allows the game to be broken by time travel. An adversary can sign a message of their choice, and then rewind time to make the game forget that it ever signed that message.

Claim 5.4. Given any signature scheme $\mathcal{S} = (\mathcal{PK}, \mathcal{SK}, \mathcal{M}, \mathcal{C}, \Sigma, \text{Gen}, \text{Sign}, \text{Verify})$, and message $m \in \mathcal{M}$, there exists an adversary $\mathcal{A}$ such that:

$$\epsilon(\mathcal{A} \circ \text{Rewind-1}(\text{EUF-CMA}_b(\mathcal{S}))) = 1$$

The idea is that the adversary can obtain a signature for m via the oracle `Sign`, and then rewind time to make the game forget that it signed this message, allowing it to query `Win`.

More formally, the following strategy always succeeds:

$$\begin{aligned}
&\sigma \leftarrow \texttt{Sign}(m) \\
&\texttt{Rewind}() \\
&\texttt{return Win}(m, \sigma)
\end{aligned}$$

By the correctness property for signatures, the σ returned by `Sign` will successfully verify. Additionally, after `Rewind`$()$, the set signed will be empty, allowing the adversary to successfully query `Win`.

■

On the other hand, UUF-CMA remains secure. This is because the message m that the adversary needs to sign is fixed after initializing the game, which means that time travel doesn't help, because no state is modified in the game.

Claim 5.5. For any signature scheme $\mathcal{S}$, $\text{Fork}(\text{UUF-CMA}(\mathcal{S})) \leq \text{UUF-CMA}(\mathcal{S})$.

Looking at the definition in Game 9, it's clear that the associated abstract game is *stateless*. We can thus apply Claim 5.1 to obtain our result.

■

6 Further Work

In this section, we look at some unanswered questions and possible lines of improvement for this work.

While we've considered a handful of different models for time travel, along with restrictions for these models, we expect that further work might develop more powerful models, or more fine-grained restrictions.

Another interesting line of work would be to investigate which classes of game make certain models of time travel equivalent. Similar to Claim 5.1 for stateless games, it might be possible to show that certain classes of games make Fork and Fork-Stack equivalent, for example.

In Claim 3.4, we separated Fork from Fork-Stack, assuming the existence of secure pseudo-random functions. One natural extension of this work would be to find the weakest primitive necessary to separate the two transformations, or even prove that a separation exists without any cryptographic assumptions.

In this work, we've considered standalone security, using games. Further work might extend models of travel to work with simulation based security, or even universally composable security [Can01]. Unfortunately, we expect that modeling time travel in a concurrent setting will prove substantially more difficult than modeling it in the standalone setting. On the other hand, this model would more likely match realistic deployments of time machines.

7 Conclusion

In this work, we investigated some impacts of time travel on cryptography.

We did this by first defining the notion of *abstract games*, allowing us to represent all security games using a common interface. We then defined various models of time travel as transformations over abstract games. We showed that these models formed a hierarchy, with backwards time travel being more powerful than no time travel, and the addition of forwards time travel being even more powerful.

We also looked at the concrete effects of our models of time travel on various cryptographic schemes. We proved that for *stateless* games, time travel provides no advantage. This led us to conclude that the IND-CPA and UUF-CMA games for encryption and signatures remained secure under time travel, and yet IND-CCA and EUF-CMA were broken, even under the weakest model of time travel.

References

[BDLF+18] Chris Brzuska, Antoine Delignat-Lavaud, Cedric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. Cryptology ePrint Archive, Paper 2018/306, 2018. `https://eprint.iacr.org/2018/306`.

[Bru03] Todd A. Brun. Computers with closed timelike curves can solve hard problems efficiently. *Foundations of Physics Letters*, 16(3):245–253, 2003.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Paper 2016/116, 2016. `https://eprint.iacr.org/2016/116`.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17:281–308, 1988.

[IVV20] Vincenzo Iovino, Serge Vaudenay, and Martin Vuagnoux. On the effectiveness of time travel to inject covid-19 alerts. Cryptology ePrint Archive, Paper 2020/1393, 2020. `https://eprint.iacr.org/2020/1393`.

[Mei22] Lúcás Críostóir Meier. State-separable proofs for the curious cryptographer. `https://cronokirby.com/posts/2022/05/state-separable-proofs-for-the-curious-cryptographer/`, 2022.

[NY90] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the 22nd annual ACM symposium on Theory of Computing*, pages 427–437, 1990.

[Yil09] Scott Yilek. Resettable public-key encryption: How to encrypt on a virtual machine. Cryptology ePrint Archive, Paper 2009/474, 2009. `https://eprint.iacr.org/2009/474`.

A Additional Game Definitions

A.1 Encryption

An encryption scheme $\mathcal{E}$ consists of types $\mathcal{K}, \mathcal{M}, \mathcal{C}$, along with functions $E : \mathcal{K} \times \mathcal{M} \xleftarrow{R} \mathcal{C}$ and $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$. By $\mathcal{M}(|m|)$ we denote the distribution of messages with the same length as m.

The encryption scheme must satisfy a correctness property:

$$\forall k \in \mathcal{K},\ m \in \mathcal{M}.\ P[D(E(k, m)) = m] = 1$$

Encrypting and then decrypting a message should return that same message.

The security of an encryption scheme can be captured by one of the following two games:

IND-CPA$_b$

$k \xleftarrow{R} \mathcal{K}$

$\underline{\texttt{Challenge}(m):}$	$\underline{\texttt{Enc}(m):}$
$m_1 \xleftarrow{R} \mathcal{M}(\lvert m_0 \rvert)$	$\texttt{return}\ E(k, m)$
$c \leftarrow E(k, m_b)$	
$\texttt{return}\ c$	

Game 7: IND-CPA$_b$

IND-CCA$_b$

$k \xleftarrow{R} \mathcal{K}$

$S \leftarrow \emptyset$

$\underline{\texttt{Challenge}(m_0):}$	$\underline{\texttt{Enc}(m):}$
$m_1 \xleftarrow{R} \mathcal{M}(\lvert m_0 \rvert)$	$\texttt{return}\ E(k, m)$
$c \leftarrow E(k, m_b)$	$\underline{\texttt{Dec}(c):}$
$S \leftarrow S \cup \{c\}$	$\texttt{assert}\ c \notin S$
$\texttt{return}\ c$	$\texttt{return}\ D(k, c)$

Game 8: IND-CCA$_b$

A.2 Signatures

A signature scheme $\mathcal{S}$ consists of types $\mathcal{PK}, \mathcal{SK}, \mathcal{M}, \mathcal{C}, \Sigma$, along with functions:

$$\begin{aligned}
&\text{Gen} : () \xrightarrow{R} \mathcal{SK} \times \mathcal{PK} \\
&\text{Sign} : \mathcal{SK} \times \mathcal{M} \xrightarrow{R} \Sigma \\
&\text{Verify} : \mathcal{PK} \times \mathcal{M} \times \Sigma \to \{0, 1\}
\end{aligned}$$

For correctness, we require that all signatures produced with a given key will successfully verify. More formally, for any message m, the following procedure always succeeds:

$$\begin{aligned}
&(\text{sk}, \text{pk}) \xleftarrow{R} \text{Gen}() \\
&\sigma \leftarrow \text{Sign}(\text{sk}, m) \\
&\texttt{return}\ \text{Verify}(\text{pk}, m, \sigma)
\end{aligned}$$

We consider two notions of security for signature schemes: UUF-CMA and EUF-CMA, with the former being weaker.

UUF-CMA_b	
$(\text{sk}, \text{pk}) \xleftarrow{R} \text{Gen}()$	$\underline{\texttt{Challenge}()}$
$m \xleftarrow{R} \mathcal{M}$	$\texttt{return}\ m$
$\underline{\texttt{Win}(\sigma) :}$	$\underline{\texttt{Sign}(\hat{m}) :}$
$\texttt{assert}\ \text{Verify}(\text{pk}, m, \sigma)$	$\texttt{assert}\ \hat{m} \neq m$
$\texttt{return}\ b$	$\texttt{return}\ \text{Sign}(\text{sk}, \hat{m})$

Game 9: UUF-CMA_b

$\boxed{\text{EUF-CMA}_b}$

$(\text{sk}, \text{pk}) \xleftarrow{R} \text{Gen}()$

$\text{signed} \leftarrow \emptyset$

$\underline{\texttt{Win}(m, \sigma):}$

$\quad \texttt{assert}\ \ m \notin \text{signed}$

$\quad \texttt{assert}\ \ \text{Verify}(\text{pk}, m, \sigma)$

$\quad \texttt{return}\ \ b$

$\underline{\texttt{Sign}(m):}$

$\quad \text{signed} \leftarrow \text{signed} \cup \{m\}$

$\quad \texttt{return}\ \ \text{Sign}(\text{sk}, m)$

Game 10: EUF-CMA$_b$