On Security Against Time Traveling Adversaries

Lúcás Críostóir Meier
lucas@cronokirby.com

September 2, 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 $G_0 \approx G_1$.

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.

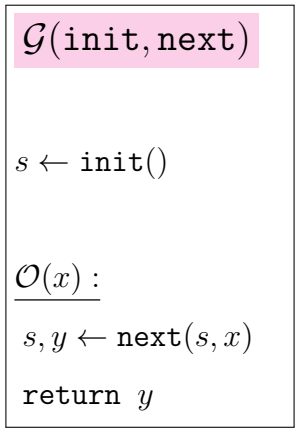
The key observation here is that the state of a pair of game is modified in only two places:

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

3.1 Rewinding Models

3.2 Forking Models

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.