Foundations and Applications for Secure Triggers

Ariel Futoransky[1] Emiliano Kargieman[1,2] Carlos Sarraute[1]
Ariel Waissbein[1,2] *

January 16th, 2005

Abstract

Imagine there is certain content we want to maintain private until some particular event occurs, when we want to have it automatically disclosed. Suppose furthermore, that we want this done in a (possibly) malicious host. Say, the confidential content is a piece of code belonging to a computer program that should remain ciphered and then "be triggered" (i.e., deciphered and executed) when the underlying system satisfies a preselected condition which must remain secret after code inspection. In this work we present different solutions for problems of this sort, using different "declassification" criteria, based on a primitive we call *secure triggers*. We establish the notion of secure triggers in the universally-composable security framework of [Canetti 2001] and introduce several examples. Our examples demonstrate that a new sort of obfuscation is possible. Finally, we motivate its use with applications in realistic scenarios.

1 Introduction

Fix a bitstring, that we regard as a secret. Let be given a family of predicates, and secretly draw a predicate from this family according to a known distribution. Think of predicates as functions with range in $\{\texttt{true}, \texttt{false}\}$. We consider algorithms that return the secret if their input evaluates to `true` on the chosen predicate, else they return nothing. Such algorithms are called *triggers*. A trigger is called *secure* if it is infeasible for an adversary, given a description of the family of predicates, the distribution used to

*[1] Corelabs, Core Security Technologies. Florida 141 (C1005AAC). Buenos Aires, Argentina.
[2] Doctorado en Ingeniería Informática, Instituto Tecnológico de Buenos Aires (ITBA). Av. Eduardo Madero 399 (C1106ACD). Buenos Aires, Argentina.

draw predicates, and the trigger's code, to recover any semantic information content of the secret.

Secure triggers have applications in malicious host problems ([Hoh98]) and software protection. Two areas of computer security that are closely related, as we shall argue, and are in need of solutions (see, e.g., [CPV03], [vO03]). This introduction to secure triggers is devoted to providing a framework for analyzing them and motivating their use in these areas.

We start our description with the *simple trigger*. For $k \in \mathbb{Z}$, a fixed security parameter, simple triggers are defined by the family of predicates $\{p_{\mathtt{b}} : \{0,1\}^k \to \{0,1\}; \mathtt{b} \in \{0,1\}^k\}$, where an element $p_{\mathtt{b}}$ is defined by the rule $p_{\mathtt{b}}(x) := \mathtt{true}$ if $x = \mathtt{b}$ and else $p_{\mathtt{b}}(x) := \mathtt{false}$. The simple trigger has been used traditionally for bootstrapping a secret, for example, in the case of a self-decryptable file that can only be decrypted by a party holding the *key*[1]. We shall construct secure implementations of this trigger (Section 4.1). However, our interest in the simple trigger is marginal —though it makes a good introductory example. Our interest lies in investigating further predicate families in order to provide a catalog of secure triggers for applications, and showing how to profit from this catalog in realistic applications.

We choose the universally-composable security (UCS) framework ([Can01]) to model secure triggers. Intuitively, UCS-secure protocols emulate *ideal processes* where parties interact with a trusted third party, the *ideal functionality*, that receives the input from the parties, makes the necessary computations, and hands them the output. In the case of secure triggers, we want the ideal functionalities to, after a successful setup, answer with the secret if and only if the input satisfies the trigger predicate. In the next section, we will give more details on this framework.

The ideal functionality for a secure trigger instance gets defined by two parameters: a family of predicates and a sampling algorithm. In practice, we will select predicates arbitrarily, hence the sampling algorithm models the attacker's *a priori* information on the predicate selection process (see, for example, Section 5).

In this work, we start a catalog of secure trigger instances with a few examples that capture the intuition of secure triggers. In each case we instantiate the secure trigger ideal functionality with a family of polynomial-time computable predicates and a polynomial-time sampling algorithm, to then

[1]Notice that the "immediate" implementation (see Section 2) of the simple trigger that *given* $H(\mathtt{b})$ *and* $\mathtt{Enc}_{\mathtt{b}}(S)$*, decrypts the encrypted secret using the input* x *as a key, i.e.,* $\mathtt{Dec}_x(\mathtt{Enc}_{\mathtt{b}}(S))$*, if the input's hash value,* $H(x)$*, agrees with the key's hash value,* $H(\mathtt{b})$*,* is not *a priori* secure under standard cryptographic assumptions (e.g., H is one-way function).

describe a protocol securely realizing this instantiated ideal functionality. Let $k \in \mathbb{Z}$ be the security parameter.

- The *simple trigger*, defined by instantiating the secure trigger ideal functionality with the family of predicates $\{p_{\mathtt{b}} : \mathtt{b} \in \{0,1\}^k\}$, where $p_{\mathtt{b}}(x) := \mathtt{true}$ if and only if $x = \mathtt{b}$, and the sampling of $\mathtt{b}$ is done uniformly in $\{0,1\}^k$.

- The *subsequence trigger* is our first non-trivial example. It allows a size parameter $s \in \mathbb{Z}$, with $s > k$, and is defined by instantiating the ideal functionality with the family of predicates $\{p_K : \{0,1\}^s \to \{\mathtt{true}, \mathtt{false}\}\}$, varying over the sets $K = \{(i_1, b_1), \ldots, (i_k, b_k)\} \subset \{1, 2, \ldots, s\} \times \{0,1\}$ such that $i_\ell \neq i_{\ell'}$ holds, for all $\ell \neq \ell'$. A predicate p_K in this family is defined by the rule $p_K(x) := \mathtt{true}$ if $x_{i_\ell} = b_\ell$ for all ℓ with $1 \leq \ell \leq s$, and else $p_K(x) := \mathtt{false}$. Sampling is done uniformly.

- The *multiple-strings trigger* allows a size parameter. For an integer s, with $s > 1$, we instantiate the ideal functionality (of size s) with the family of predicates $\{p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}} : \mathtt{b}^{(j)} \in \{0,1\}^k, \forall j = 1, \ldots, s\}$, where the predicate $p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}}$ evaluates to $\mathtt{true}$ on input $x \in \{0,1\}^*$, of size $|x|$, if there exist indices $1 \leq i_1, \ldots, i_s \leq |x| - k + 1$ such that $(x_{i_j}, \ldots, x_{i_j+k-1}) = \mathtt{b}^{(j)}$ for every j, with $1 \leq j \leq s$. Sampling is done uniformly in $\{0,1\}^{sk}$.

These triggers have immediate applications, as we show in Sections 1.1 and 5. However, the job of broadening this catalog remains. One of our main aims is to study the extent of secure triggers, particularly in the cases of malicious host problems and software protection. In this sense, our study of sampling algorithms is left at an introductory level (see Remark 4.4), and thus, the design for applications will require much caution (see, for example, Section 5).

Some other problems that we analyzed were left open. We are particularly interested in what we called the *finite-state-machine trigger*. That is, a finite-state machine such that both its states and transitions are (probabilistically) encrypted and the machine runs in this ciphered form until the "trigger state" is reached, when all information is automatically deciphered. It remains to see if one such secure trigger exists. Applications of this trigger include software protection and remote pattern matching.

1.1 Motivation

The malicious host problem deals with securing the execution of trusted mobile code run on untrusted hosts. Solutions to these problem based on secure triggers hide "sensitive" functionalities as "completely obfuscated" code until required for use. The mobile code is deployed on this host after a setup stage. Once deployed, the mobile code processes the data in this host's memory —for example, scans databases or maybe is fed by input sent to these servers—, and provides the embedded trigger with inputs. When an input satisfies the underlying predicate, the obfuscated (i.e., encrypted) functionalities are deciphered and executed. Hence, malicious hosts can neither tamper with nor determine these functionalities before they are triggered (though, they can be blocked).

The following application examples show possible uses of secure triggers in the malicious host problem setting. The first one shall be revisited in detail in Section 5.

Anonymous shopping agent: Some gentleman wants to buy a very rare product (make/model/color/price) using a mobile-code program that crawls some etailers' websites. Given the nature of his business, he requires that no other party can find out his identity, what is he looking for, or how much money is he prepared to pay (except for the etailer where he will make the buy). He thus implements a secure trigger —to be deployed in several etailers servers as anonymous mobile code— that searches for this target product and then, only when it is found, the secure trigger produces his name an emails himself the details. The trigger produces the sensitive information only when needed. Moreover, the only parties *finding out* this information are the etailers that have the target product in their sites. Assuming that the range of products sold world wide by etailers is large enough, we can guarantee that brute-force attacks to this trigger become infeasible.

More generally, this problem relates to private-information retrieval. Under certain assumptions on databases, the user is able to search databases for a target entry (described partially by some of its attributes) while no attacker examining or controlling a database missing the target entry can infer it from the mobile code.

Information warfare worm attack: Anticipating an international-conflict scenario, the information-warfare department of one of the nations involved anonymously disseminates a worm over the Internet. The worm looks inconspicuous enough: takes control of a machine, records keystrokes, steals passwords, and tries to infect other systems.

As systems are infected, an embedded trigger is fed with configuration information used to detect potential targets. To make it resistant to brute-force attacks, only a small portion of the target parameters satisfy the trigger predicate. For example, assuming that there is a sufficiently large set of possible configurations, we can use the subsequence trigger to check if certain bits of a description match those same bits in the target description without leaking them. When a matching target is found, the trigger decrypts and executes a sophisticated module that scans the machine's hard drives for sensitive information, compresses, encrypts and uses a steganographic channel to transmit the information back to the worm's creators.

None of this specialized behavior can be inferred by inspecting the worm's code (until it triggers): The security team of the nation under attack cannot answer "What is the worm looking for?" nor "What does the worm do after finding it?".

Our interest in secure triggers grew from research done by the authors and others in the design of a practical software protection tool ([BFN+03]). The goal of this framework is to enforce license policy and embed robust watermarks. Its basic ingredient, is obfuscation through secure triggers. The software protection application is very complex in itself and its description lies beyond the scope of this work. For a deeper discussion on this, we refer to [BFN+03]. We will briefly get back to other obfuscation issues in Section 2.

2 Related work

A few protocols similar to certain trigger instances have already been published, while most of them are only related to the simple trigger. In this sense it is worth pointing out the paper [JS02], where authors introduce *fuzzy vaults*: a scheme that reconstructs an encrypted secret from a set of shares permitting a small portion of them to be corrupted (e.g., the secret can be re-constructed with any eighteen out of twenty shares). Fuzzy vault is an example of secure trigger that complements our catalog.

Simple triggers have been used for the construction of virus. In the early 90s a virus called Cheeba searched in the file-system of infected machines for a specific file (`USERS.BBS`) to launch the virus program ([Per03]). The search was done in a trigger-like manner by matching the hash value of each of the infected computer's files with a "hard-coded" value (see [Gry92]). A post in the Slashdot webpage comments on a worm with this behavior [Ano02][2].

[2]These applications hint on our cyber-warfare application described earlier in this sec-

More generally, the simple trigger can be related to well-known cryptographic protocols such as: password authentications —checked by comparing the candidate's digest to the original password's digest— before decryption; oracle hashing ([Can97, Can00b, CMR98]); commitment ([Blu81]); all-or-nothing transforms ([Riv97]); etcetera. These constructions agree on one thing, after a secret value is fed they disclose a secret while they remain secure against offline attacks. We remark that these notions and secure triggers are essentially distinct. Primarily, because secure triggers permit arbitrary predicate families, other than the simple trigger, while it is not apparent if any one of these primitives could be used to construct other instances secure triggers. Another difference can be spotted when comparing these notions at protocol level, since the party intended to "hit" the trigger in the execution of a trigger protocol is not necessarily that who sets up the trigger procedure (and knows the secret), but external parties which hold the information satisfying the trigger predicate. Indeed, simple triggers are not commitment schemes, since the party doing the setup for a secure trigger (e.g., the commitment) is not required to open the secret (e.g., decommit)[3]. (Compare with protocol $\pi_{\mathtt{simple-a}}$ in Section 4.1.) Conversely, it is not the focus of this work to give further implementations of the simple trigger.

Other primitives worth mentioning include those coming from timed-release cryptography (e.g., [DN93], [RSW96], [DOR99] and [BN00]), where the goal is to maintain secrecy until a predetermined amount of time has passed. But, with secure triggers the goal is to maintain secrecy until a predetermined event occurs.

Finally, it is important to compare our results with the recent obfuscation results ([BGI$^+$01], [LPS04]). One can argue that, in order to achieve a reasonable form of software protection in todays computers (e.g., without requiring special hardware), obfuscation is required. For example, if a program includes license checks that can be analyzed and tampered with by a skilled programmer, then pirates will probably be able to thwart these license checks. From a cryptographic standpoint, one cannot design an obfuscation algorithm (for sufficiently general Turing machines) as defined and proved in [BGI$^+$01]. Let us mention that the "trigger obfuscation" is of a different nature, as secure triggers can only be used to obfuscate very special Turing machines; "trigger obfuscation" blocks analysis before the code is executed —but not after.

tion.

[3] In particular, simple triggers are not UC commitment schemes and the impossibility result of [CF01] cannot be applied.

As a side note, let us mention that this new paradigm, proved to be very useful in the construction of a software protection framework ([BFN+03]). The key idea here, is to have certain portions of the code —possibly containing license checks— obfuscated with triggers. Each of these portions of code is embedded in a trigger (as a secret), and the predicate is selected according to the value held by certain local variables, in an untampered run of the program, at the instant immediately before the portion of code must be executed. Commercial software is complex (branching) and will typically include several portions of code that are executed under very stringent conditions, e.g., rare events are described by rare values of the local variables, it happens that it is difficult for an attacker to infer the value of these variables (and hit the trigger). See more on this in *op. cit.*.

In [LPS04], authors adapt the definition of obfuscation of [BGI+01] to the random oracle model in order to design a family of algorithms, *point-functions with general output*, that is obfuscatable. This functions and the simple trigger ideal functionality have the same input/output behavior!

3 The Universally Composable Security framework

By now, Canetti's UCS framework ([Can00a, Can01]) has become popular in cryptology, as many authors analyze the security of protocols using it. The best reference is Canetti's own paper [Can00c] (see also [CF01], [CK02], [DN02]).

The UCS framework aims to capture the task of secure function evaluation in an asynchronous, ideally authenticated network. To this end two models are considered, a first model that represents a protocol execution in real life and a second model that captures the security requirements of the given task in an idealized setting. A protocol is said secure in the UCS sense if no *interactive distinguisher* can tell apart an execution in the real-life model from one in the idealized setting.

Explicitly, in both worlds the parties are interactive, probabilistic, polynomial-time Turing machines (ITM for short). All parties participate in a message-driven protocol inside an asynchronous network without guaranteed delivery of messages, where communication is public and unauthenticated. These parties interact with two adversarial parties called *environment* (the interactive distinguisher) and *adversary* —also ITMs.

- Real-life protocol: parties interact during the execution of the protocol communicating through channels accessible to the attacker (for eavesdropping, stopping, or inserting messages). The parties receive their

input directly from the environment, through special I/O channels inaccessible to the attacker, and work out the result by themselves. The output of the protocol is forwarded to the environment.

- Ideal process: A protocol execution amounts to the environment delivering the input to *dummy* parties that make no computations and forward their input to a trusted party, the *ideal functionality*, an additional ITM which remains unseen by the environment. This party makes all the necessary protocol calculations and returns them with the result, which they output for the environment. In this scenario, the attacker cannot eavesdrop on communications, but can freely communicate with the ideal functionality; it can only detect and stop a message from the ideal functionality to the parties (but cannot see its contents).

The environment acts as an interactive distinguisher: over a coin toss (its result being secret to the environment), it will participate in the execution of a real-life protocol or an ideal process. It will provide parties with their input and witness their outputs. It will communicate with the adversary arbitrarily. Once the protocol execution is finished, it must decide whether it has participated in a real-life protocol or in an ideal process. A protocol is said to *securely realize* an ideal functionality with respect to a class of adversaries $\mathcal{C}$ if for every *real-life adversary* in $\mathcal{C}$, there exists an *ideal-process adversary* in $\mathcal{C}$ such that no environment can tell whether it participates in a real-life protocol or in an ideal process with non-negligible probability. We consider two settings: static adversaries and adaptive adversaries. In the adaptive setting, as in [CK02], we will allow protocol parties to erase local data.

4 Secure trigger procedures

We want to model trigger algorithms as non-interactive algorithms that do not leak semantic information for the secret nor information reveal the selected predicate, when the setup occurs without active intervention from the attacker. This is captured by the secure trigger ideal functionality described below.

The following assumption will be used throughout the paper. Ideal functionalities do not accept secrets of size smaller than the security parameter. The condition is not required by all our protocols, but will be assumed

Functionality $\mathcal{F}$

$\mathcal{F}$ proceeds as follows running with parties D, T and adversary $\mathcal{S}$.

1. Wait for a message from D of the form $(\texttt{Setup}, S)$ and record S. Put the message TriggerActivated in the outgoing communication tape with recipient T, and send $(\mathsf{Setup}, |S|)$ to $\mathcal{S}$.

2. Wait for one of the messages $(\mathsf{Trigger}, \mathcal{F})$ or $(\mathsf{Trigger}, \mathcal{S})$ from $\mathcal{S}$ and record this message.

   a) If the message was $(\mathsf{Trigger}, \mathcal{F})$, use Samp to draw a predicate p from $\{p\}$. Next, For any input (Check, x) received from T, if $p(x) = \texttt{true}$ holds, return the secret to T. Else, do nothing.

   b) If it was $(\mathsf{Trigger}, \mathcal{S})$, hand any input (Check, x) received from T to $\mathcal{S}$. If $\mathcal{S}$ answers this message, forward the answer to T; else, do nothing.

Figure 1: The Secure Trigger Ideal Functionality

throughout for the sake of simplicity. Note that it is not particularly restrictive in practice, e.g., for typical choices of security parameter and secret.

Formally, the environment interacts with an adversary and two protocol parties: a dealer, D, and a triggerer, T. A secure trigger ideal functionality instance is parametrized by a family of polynomial-time computable predicates $\{p\}$ and a polynomial-time sampling algorithm Samp (supported on $\{p\}$). An instance $\mathcal{F} := \mathcal{F}^{(\{p\},\mathsf{Samp})}$ expects to receive a message of the form (Setup, S) from D. If the size of the secret, $|S|$, is smaller than the security parameter k, it terminates. Else, it uses Samp to draw a predicate p from the given family of predicates, it forwards the message $(\mathsf{Setup}, |S|)$ to the ideal-process adversary, and sends the output TriggerActivated to T. At any time, the ideal functionality waits for one of the messages $(\mathsf{Trigger}, \mathcal{S})$ or $(\mathsf{Trigger}, \mathcal{F})$ from the ideal-process adversary. In the first case, for every input (Check, x) received from T, $\mathcal{F}$ will output S for the triggerer if $p(x) = \texttt{true}$. In the second case, the ideal functionality will forward every input (Check, x) to the ideal-process adversary and return T with whatever $\mathcal{S}$ answers. Else it does nothing. This is summarized in Figure 1.

Step 2 in Figure 1 might require some explanation. Indeed, we will assume in practice that the setup (Step 1) is done without active participation of the adversary. This means in particular that no attacker can tamper with

the setup message (in which case $\mathcal{S}$ answers $(\mathsf{Setup}, \mathcal{F})$, thus simplifying the ideal process). We do not require this here and make no further assumptions. We could, of course, have the setup message authenticated; but this would also require unnecessary preliminary work. Instead, we allow the ideal functionality to "change plans" if faced with a real-life adversary that tampers with the setup message sent by D to T (Fig. 1, (2.b)). The ideal-process adversary will detect if the real-life adversary tampers with the setup message and interact with the ideal functionality so that the tampering does not allow the environment to distinguish one protocol from the other.

In order to design real-life protocols for secure triggers we might require the use of certain cryptographic primitives: a ind-cpa secure symmetric cipher $(\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$, a one-way function $h : \{0,1\}^* \to \{0,1\}^*$, and a collection of pseudo-random generators. In the latter case, for integers $k, s \in \mathbb{Z}$ with $s \geq k$, we shall denote by $G_{k,s}$ a pseudo-random generator that expands strings of size k to strings of size $k + s$. See for instance [Gol01, Gol04].

4.1 The simple trigger

Let $k \in \mathbb{Z}$. The ideal functionality for a simple trigger, $\mathcal{F}_{\mathtt{simple}}$, with security parameter k is defined by the predicate family

$$\left\{ p_{\mathtt{b}} : \{0,1\}^* \to \{\mathtt{true}, \mathtt{false}\}; \mathtt{b} \in \{0,1\}^k \right\},$$

where $p_{\mathtt{b}}(x) := \mathtt{true}$ if $x = \mathtt{b}$, else $p_{\mathtt{b}}(x) := \mathtt{false}$; together with the sampling algorithm that selects $\mathtt{b} \in \{0,1\}^k$ according to the first k bits of its random tape. We shall design two protocols, one securely realizing $\mathcal{F}_{\mathtt{simple}}$ with respect to static adversaries ($\pi_{\mathtt{simple-s}}$) and another one securely realizing it with respect to adaptive adversaries when local erasure of data is allowed ($\pi_{\mathtt{simple-a}}$). The former is concise, easier to understand, and might be enough for most applications. The latter, presents a more delicate problem and requires a less direct solution.

Protocol $\pi_{\mathtt{simple-s}}$: The dealer waits for an input of the form (Setup, S) and terminates if $|S| < k$ holds. Else, D uses the key generation algorithm $\mathtt{Gen}$ to generate $\mathtt{b} := \mathtt{Gen}(1^k)$, writes the message $(\mathtt{Setup}, \mathtt{Enc}_{\mathtt{b}}(0^k), \mathtt{Enc}_{\mathtt{b}}(S))$ in its outgoing communication tape with recipient T, and terminates. The triggerer expects a message of the form (Setup, A, B), for $A, B \in \{0,1\}^*$, and outputs TriggerActivated. Then, for every input $(\mathtt{Check}, x)$ that T receives, it will output $\mathtt{Dec}_x(B)$, if $\mathtt{Enc}_x(0^k) = A$. Else, it does nothing.

Theorem 4.1 *Protocol* $\pi_{\mathtt{simple-s}}$ *securely realizes* $\mathcal{F}_{\mathtt{simple}}$ *with respect to static adversaries.*

Proof: Fix a real-life adversary. We construct an ideal-process adversary such that no environment $\mathcal{Z}$ can distinguish if it participate in a real-life protocol or an ideal-process execution with more than negligible probability.

The ideal-process adversary, $\mathcal{S}$, we construct does the following. In parallel to the ideal-process execution, it simulates a virtual copy, $\mathcal{A}$, of the real-life adversary in a black-box way. It imitates a copy of the execution of the real-life protocol $\pi_{\mathtt{simple-s}}$ for $\mathcal{A}$, and forwards all messages from $\mathcal{Z}$ to $\mathcal{A}$ and back. More explicitly, the ideal process behaves as follows.

Assume that no party is corrupted. When $\mathcal{S}$ receives the size of the secret, $|S|$, from the ideal functionality, it generates $\mathtt{b}' := \mathtt{Gen}(1^k)$ and simulates the message $(\mathtt{Setup}, \mathtt{Enc}_{\mathtt{b}'}(0^k), \mathtt{Enc}_{\mathtt{b}'}(1^{|S|}))$ for $\mathcal{A}$ with sender D and recipient T. Here 0^k and $1^{|S|}$ denote the all-zeroes string of size k and the all-ones string of size $|S|$, respectively.

At any time $\mathcal{S}$ waits for $\mathcal{A}$ to send a setup message to T, which we denote by (Setup, A, B), and lets through all messages from the ideal functionality to the triggerer. If this setup message agrees with that simulated by $\mathcal{S}$, i.e., if $\mathtt{Enc}_{\mathtt{b}'}(0^k) = A$ and $\mathtt{Enc}_{\mathtt{b}'}(1^{|S|}) = B$, then the ideal-process adversary sends the message $(\mathsf{Trigger}, \mathcal{F}_{\mathtt{simple}})$ to the ideal functionality. Else, it sends the message $(\mathsf{Trigger}, \mathcal{S})$; next, for every message (Check, x) it receives from $\mathcal{F}_{\mathtt{simple}}$, it returns $\mathtt{Dec}_x(B)$ if $\mathtt{Enc}_x(0^k) = A$; else, it does nothing. This finishes the description of $\mathcal{S}$ in the uncorrupted case.

Assume that some party is corrupted, and recall that in the static setting corruption occurs on startup. Then, $\mathcal{S}$ corrupts the corresponding party, provides $\mathcal{A}$ with the internal state of the corrupted party, and follows $\mathcal{A}$'s instructions. If the dealer was corrupted, then $\mathcal{S}$ will also deliver S to $\mathcal{A}$. Notice that $\mathcal{S}$ needs to imitate the behavior of the corrupted parties for $\mathcal{A}$, for example, if $\mathcal{A}$ corrupts the dealer and instructs it to run according to its algorithm, then $\mathcal{S}$ will compute and deliver the setup message $(\mathtt{Setup}, \mathtt{Enc}_{\mathtt{b}}(0^k), \mathtt{Enc}_{\mathtt{b}}(S))$ to $\mathcal{A}$, with sender D and receiver T, for some $\mathtt{b} = \mathtt{Gen}(1^k)$.

The proof is completed by showing that $\mathcal{Z}$ cannot use the differences between the two runs to distinguish one from the other. But there is only one difference[4], and it happens in the uncorrupted case: the environment receives the string $\mathtt{Enc}_{\mathtt{b}'}(0^k), \mathtt{Enc}_{\mathtt{b}'}(1^{|S|})$ from the adversary in the ideal process, while it receives $\mathtt{Enc}_{\mathtt{b}}(0^k), \mathtt{Enc}_{\mathtt{b}}(S)$ in the real-life protocol. It is easy to see that an environment that distinguishes one from the other with non-negligible probability, breaks the given ind-cpa secure encryption scheme.

Protocol $\pi_{\mathtt{simple-s}}$ is insecure against adaptive adversaries, this is easy to see because if the real-life adversary corrupts the dealer after delivering the setup message, it retrieves both $\mathtt{b}$ (or $\mathtt{b}'$) and S. However, at that point of the protocol execution, the ideal-process adversary has already delivered $(\mathsf{Setup}, \mathtt{Enc}_{\mathtt{b}'}(0^k), \mathtt{Enc}_{\mathtt{b}'}(1^{|S|}))$ which is inconsistent with $\mathtt{b}'$ and S. That is, the value $\mathtt{Enc}_{\mathtt{b}'}(1^{|S|})$ binds $\mathcal{S}$ to $1^{|S|}$. In order to construct a protocol secure with respect to adaptive adversaries, we will use one-time pad encryption (compare with [CK02, Section 5.1]).

Protocol $\pi_{\mathtt{simple-a}}$: The dealer waits for an input of the form (Setup, S) and terminates if $|S| < k$ holds. Else, D uses a pseudo-random generator to expand k bits from its random tape, $\mathtt{b}$, to a bitstring $(\mathtt{b}^{(1)}, \mathtt{b}^{(2)}) := G_{k,|S|}(\mathtt{b})$ in $\{0,1\}^k \times \{0,1\}^{|S|}$, and deletes the value of $\mathtt{b}$ from its memory. Then, the dealer puts the message $(\mathsf{Setup}, \mathtt{b}^{(1)}, \mathtt{b}^{(2)} \oplus S)$[5] in its outgoing communication tape with recipient T and terminates. The triggerer expects a setup message (Setup, A, B), for some $A, B \in \{0,1\}^*$, and outputs $\mathsf{TriggerActivated}$. Next, for every input $(\mathtt{Check}, x)$ it receives, it computes $(x^{(1)}, x^{(2)}) := G_{k,|S|}(x)$, and if $x^{(1)} = A$, it outputs $x^{(2)} \oplus B$. Else it does nothing.

For $s \geq k$, consider $G_{k,s}$ as a function with range in $\{0,1\}^k \times \{0,1\}^s$. Notice that by [Gol01, Proposition 3.3.8], the map $\{0,1\}^k \to \{0,1\}^k$ defined by the rule $x \mapsto x^{(1)}$, for $(x^{(1)}, x^{(2)}) = G_{k,s}(x)$, defines a one-way function. Therefore, if $(x^{(1)}, x^{(2)}) := G_{k,s}(x)$, $(\mathtt{b}^{(1)}, \mathtt{b}^{(2)}) := G(\mathtt{b})$, and $x_1 = \mathtt{b}^{(1)}$, then $x^{(2)} = \mathtt{b}^{(2)}$ holds except with negligible probability. We are ready to prove the following theorem.

Theorem 4.2 *Protocol $\pi_{\mathtt{simple-a}}$ securely realizes $\mathcal{F}_{\mathtt{simple}}$ with respect to adaptive adversaries if local data erasure is allowed.*

Proof: Fix a real-life adversary. We define an ideal-process adversary $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish one from the other with non-negligible probability.

The adversary $\mathcal{S}$ runs a virtual copy of the real-life adversary $\mathcal{A}$ in parallel with the execution of the ideal process. It imitates a copy an execution of the real-life protocol $\pi_{\mathtt{simple-a}}$ for $\mathcal{A}$, and will forward all messages from $\mathcal{Z}$ to $\mathcal{A}$ and vice versa. Let the ideal process execution start, and assume for now that $\mathcal{A}$ has not corrupted the dealer yet. Then, the ideal-process

[4] Other differences occur with negligible probability and can be ignored, e.g., it might happen that $\mathtt{Enc}_x(0^k) = \mathtt{Enc}_{\mathtt{b}}(0^k)$ but $p_{\mathtt{b}}(x) = \mathtt{false}$ —but only with negligible probability.

[5] Here $\oplus$ stands for the bitwise X-OR operation.

adversary waits for a message containing the size of the secret, $|S|$, from the ideal functionality and selects the strings $c_1 \in \{0,1\}^k, c_2 \in \{0,1\}^{|S|}$ from to its random tape, and hands the message $(\mathtt{Setup}, c_1, c_2)$ to $\mathcal{A}$ with sender D and recipient T.

The ideal-process adversary waits for $\mathcal{A}$ to send a setup message (Setup, A, B) to T, for some $A, B \in \{0,1\}^*$, at any time of the protocol execution. Once sent, $\mathcal{S}$ lets through all messages from the ideal functionality to the triggerer. If this setup message agrees with the one simulated by $\mathcal{S}$, i.e., if $c_1 = A$ and $c_2 = B$, then $\mathcal{S}$ sends the message $(\mathsf{Trigger}, \mathcal{F}_{\mathtt{simple}})$ to the ideal functionality, and sends $(\mathsf{Trigger}, \mathcal{S})$ otherwise. In the latter case, for every message (Check, x) received from $\mathcal{F}_{\mathtt{simple}}$, $\mathcal{S}$ computes $(x^{(1)}, x^{(2)}) := G_{k,|S|}(x)$, and returns $B \oplus x^{(2)}$ if $x^{(1)} = A$. Else, $\mathcal{S}$ does nothing.

In the case that $\mathcal{A}$ corrupts D or T, then $\mathcal{S}$ corrupts the corresponding party and provides $\mathcal{A}$ with its internal state. If D is corrupted before $|S|$ has been received, then $\mathcal{S}$ simulates a "real" dealer for $\mathcal{A}$: it hands S along with the setup message $(\mathsf{Setup}, \mathtt{b}^{(1)}, \mathtt{b}^{(2)} \oplus S)$, with sender D and recipient T, where $\mathtt{b} \in \{0,1\}^k$ is chosen from the ideal-process adversary's random tape and $(\mathtt{b}^{(1)}, \mathtt{b}^{(2)}) := G_{k,|S|}(\mathtt{b})$. If the corruption occurs after $|S|$ was received (thus, $\mathtt{b}$ has already been deleted and $\mathcal{A}$ has received the "fake" setup message $(\mathsf{Setup}, c_1, c_2)$), then the ideal-process adversary obtains the secret S from the dealer and hands $\mathcal{A}$ the values $c_1, c_2 \oplus S$ that are consistent with the setup message and secret.

In order to show that protocol $\pi_{\mathtt{simple-a}}$ securely realizes $\mathcal{F}_{\mathtt{simple}}$, we notice that there is a single difference between the two runs. The values $\mathtt{b}^{(1)}$ and $\mathtt{b}^{(2)}$ generated by D in the real-life protocol are computationally independent, whereas the values c_1 and $c_2 \oplus S$ generated by $\mathcal{S}$ in the ideal process are independently distributed. However, one can see that breaking the security of this protocol can be reduced to breaking the security of $G_{k,|S|}$, which proves the theorem. □

4.2 The subsequence trigger

Let $s, k \in \mathbb{Z}$ be fixed integers with $s > k$. The subsequence trigger ideal functionality, $\mathcal{F}_{\mathtt{subset}}$, is defined by the family of predicates

$$\Big\{ p_K : \{0,1\}^s \to \{\mathtt{true}, \mathtt{false}\}; K = \{(i_1, b_1), \ldots, (i_k, b_k)\} \subset \{1, \ldots, s\} \times \{0,1\}, \\ \text{such that } i_\ell \neq i_{\ell'} \text{ if } \ell \neq \ell' \Big\},$$

where a predicate p_K in this family evaluates to $\mathtt{true}$ on input $x = (x_1, \ldots, x_s) \in \{0,1\}^s$, if and only if, $x_{i_\ell} = b_\ell$ holds for every ℓ with $1 \leq \ell \leq$

k. The sampling algorithm, will independently select $(b_1, \ldots, b_k) \in \{0,1\}^k$ according to its random tape and k distinct integers $i_1, \ldots, i_k$ in $\{1, \ldots, s\}$ (see Figure 2).

let $J := \{1, \ldots, s\}$;
for $\ell := 1$ **to** k **do:** {
    set b_ℓ $\{0,1\}$;
    set i_ℓ J;
    let $J := J \setminus \{i_\ell\}$;
    };
output $K = \{(i_1, b_1), \ldots, (i_k, b_k)\}$;

Figure 2: Sampling algorithm for K

We remark that according to this ideal functionality, both the sequence $i_1, \ldots, i_k$ and $\mathtt{b} = (b_1, \ldots, b_k)$ are not leaked during the protocol execution, except if the trigger is hit.

We shall design a protocol that securely realizes this ideal functionality with respect to adaptive adversaries. To implement the real-life protocol we construct an auxiliary family of (polynomially-computable) functions $\{\tau : \{0,1\}^s \to \{0,1\}^k\}$. Given $K = \{(i_1, b_1), \ldots, (i_k, b_k)\}$ as above we can produce (an algorithm for) a function τ such that $p_K(x) = \mathtt{true}$ if and only if $\tau(x) = (b_1, \ldots, b_k)$, then the triggerer will be able to decide if $p_K(x) = \mathtt{true}$ by checking if the first entry of $G_{k,s}(\tau(x))$ agrees with $\mathtt{b}^{(1)}$, for given τ and $\mathtt{b}^{(1)}$ (where $(\mathtt{b}^{(1)}, \mathtt{b}^{(2)}) = G_{k,s}(\mathtt{b})$). But with the added property that τ and $\mathtt{b}^{(1)}$ do not leak $\mathtt{b}^{(2)}$ (cf., Protocol $\pi_{\mathtt{simple-a}}$)! The family $\{\tau\}$ verifies the following properties:

1. Let be given $K = \{(i_1, b_1), \ldots, (i_k, b_k)\}$ as above and $x \in \{0,1\}^s$ such that $p_K(x) = \mathtt{true}$, then we can compute the algorithm for a function τ such that $\tau(x) = (x_{i_1}, \ldots, x_{i_k}) = (b_1, \ldots, b_k)$.

2. Every function τ in this family is onto, and maps an input x in $\{0,1\}^s$ to $\tau(x) = \tau(x_1, \ldots, x_s) = (x_{j_1}, \ldots, x_{j_k})$, for some distinct integers $j_1, \ldots, j_k \in \{1, \ldots, s\}$. Furthermore, if $y \in \{0,1\}^s$ is such that $y_{j_\ell} = x_{j_\ell}$ for $1 \leq \ell \leq k$, then both values have the same output $\tau(x) = \tau(y)$.

Now, we can see from the above properties, that the conditions "$p_K(x) = \mathtt{true}$" and "$\tau(x) = (b_1, \ldots, b_k)$" are equivalent.

With out further delays let H be a one-way function and assume that the output is of a fixed size for all input of size s, i.e., $|H(x)| = m$ for every

$x \in \{0,1\}^s$ (and some $m \in \mathbb{Z}$). Let

$$\left\{\sigma_{(t_1,\ldots,t_s)} : \{1,2,\ldots,s\} \times \{0,1\}^s \to \{0,1\}^k; t_i \in \{0,1\}^m, \text{ for } 1 \leq i \leq s\right\}$$

be a family of functions, where $\sigma_{(t_1,\ldots,t_s)}(i,x) := y := (y_1,\ldots,y_k)$ is defined by the algorithm in Figure 3. We need some notation. Let J denote the finite sequence $J := (1,\ldots,s)$. For every finite sequence of distinct integers I, and every integer i in I, let $I \setminus \{i\}$ denote the sequence obtained by deleting i and re-indexing all integers to the right of i. For example $(1,2,3,4) \setminus \{2\} = (1,3,4)$.

Stored: $(t_1,\ldots,t_s)$.
Input: $j, (x_1,\ldots,x_s)$.

let $j_1 := j; y_1 := x_j; J := (1,\ldots,s) \setminus \{j_1\}$;
for $\ell := 2$ **to** k **do:** {
    compute $n := \left(H(j_1|y_1|\ldots|j_{\ell-1}|y_{\ell-1}) \oplus t_{j_{\ell-1}}\right) \mod (s-\ell+1)$;
    let j denote the n-th entry in J;
    let $j_\ell := j; y_\ell := x_j; J := J \setminus \{j_\ell\}$;
output $(y_1,\ldots,y_k)$;

Figure 3: The auxiliary function

This algorithm is short but might not be so easy to follow. A function σ is uniquely defined by elements $t_1,\ldots,t_s$. Assume them fixed. We briefly describe how to evaluate σ in an input j_1, x for $x \in \{0,1\}^s, j_1 \in \mathbb{Z}$ with $1 \leq j_1 \leq s$. According to the algorithm (Figure 3), the first bit of the output, y_1, is defined as $y_1 := x_{j_1}$. Assume that $i_1, y_1, \ldots, i_{\ell-1}, y_{\ell-1}$ have been computed and $\ell \leq k$. To compute i_ℓ, y_ℓ we let $n := H(j_1|y_1|\cdots|j_{\ell-1}|y_{\ell-1}) \oplus t_{j_{\ell-1}}$, consider this bitstring as an integer, and compute its remainder modulo $s-\ell+1$ (the pipe in the above expression stands for the concatenation). The next line in the algorithm simply means that j_ℓ must be selected as the n–th undeleted integer from $\{1,\ldots,s\}$, counting from the left. Then, we set $y_\ell := x_{j_\ell}$.

Given any function σ, notice that for every j $(1 \leq j \leq s)$, the function

$$\tau := \sigma(j, \ ) : \{0,1\}^s \to \{0,1\}^k$$

trivially verifies property 2 above. To see that property 1 is also verified, notice that given K and $x \in \{0,1\}^s$ such that $p_K(x) = \texttt{true}$, one can run

the algorithm on input (i_1, x) without specializing the values of $t_1, \ldots, t_s$, and sequentially specialize each $t_{j_{\ell-1}}$ when required so that $j_\ell = i_\ell$. We are now ready to describe the protocol for this trigger.

Real-life protocol for the subsequence scheme ($\pi_{\mathtt{subseq-a}}$).– The dealer waits for an input of the form (Setup, S) and terminates if $|S| < k$ holds. Else, D sets $\mathtt{b} \in \{0,1\}^k$ and $t_1, \ldots, t_s \in \{0,1\}^m$ according to $k + ms$ bits of its random tape, uses the pseudo-random generator algorithm to compute $(\mathtt{b}^{(1)}, \mathtt{b}^{(2)}) := G_{k,|S|}(\mathtt{b})$, deletes $\mathtt{b}$ from its memory, writes the message $(\mathtt{Setup}, (t_1, \ldots, t_s), \mathtt{b}^{(1)}, \mathtt{b}^{(2)} \oplus S)$ in its outgoing communication tape with recipient T, and terminates. The triggerer expects a setup message of the form $(\mathtt{Setup}, R, A, B)$, for $R \in (\{0,1\}^m)^s$, and $A, B \in \{0,1\}^*$, and outputs $\mathsf{TriggerActivated}$. Next, for every input $(\mathtt{Check}, x)$ that T receives, for every $i = 1, \ldots, s$: the triggerer runs the algorithm of Figure 3 with input i, x. Denote its output by $\sigma(i, x)$, and write $(x^{(i,1)}, x^{(i,2)}) := G_{k,|S|}(\sigma(i, x))$. Then, the triggerer checks whether $x^{(i,1)} = A$, and if it does, it outputs $x^{(i,2)} \oplus B$. Else, it does nothing.

Notice that T is not given a function $x \mapsto \tau(x)$ but the collection of functions $\{x \mapsto \sigma(i, x); 1 \leq i \leq s\}$. Asymptotic computations for T remain in $(sk)^{O(1)}$.

Theorem 4.3 *Protocol $\pi_{\mathtt{subseq-a}}$ securely realizes $\mathcal{F}_{\mathtt{subseq}}$ with respect to adaptive adversaries if local erasure of data is allowed.*

Proof: The proof follows the lines of the proof of Theorem 4.2. Given a real-life adversary, we construct an ideal-process adversary $\mathcal{S}$ such that no environment can decide if it participates in a real-life protocol or in an ideal-process execution with non-negligible probability. The proof is again by simulation, and only a few modifications need to be made. That is, $\mathcal{S}$ simulates a copy of the real-life adversary $\mathcal{A}$, it functions as an interface so that $\mathcal{A}$ and the environment communicate freely, and imitates a copy of $\pi_{\mathtt{subset-a}}$ for $\mathcal{A}$.

In the case that D is not corrupted before setup, the ideal adversary simulates a setup message for $\mathcal{A}$: $(\mathsf{Setup}, (t'_1, \ldots, t'_s), c_1, c_2)$, where $t'_1, \ldots, t'_s$ are s random values selected uniformly in $\{0,1\}^m$ and c_1 and c_2 are random values in $\{0,1\}^k$ and $\{0,1\}^s$, respectively. Again, the ideal-process adversary waits for $\mathcal{A}$ to forward a setup message to the triggerer, and lets through messages from $\mathcal{F}_{\mathtt{subset}}$ to T. If the setup message was modified to $(\mathsf{Setup}, R, A, B)$ (or created), with $R \in (\{0,1\}^m)^s$ and $A, B \in \{0,1\}^*$, then the adversary sends the message $(\mathsf{Trigger}, \mathcal{S})$ to the ideal functionality.

Next, for every input (Check, x) it receives from $\mathcal{F}_{\mathtt{subset}}$, and for every i with $1 \leq i \leq s$, it computes $(x^{(i,1)}, x^{(i,2)}) := G_{k,|S|}(\sigma(i,x))$, and it returns $x^{(i,2)} \oplus B$ if $x^{(i,1)} = A$ (here $\sigma := \sigma_R$ denotes the auxiliary function defined by R according to the algorithm in Figure 3). Else, it returns nothing.

If some party is corrupted, then $\mathcal{S}$ corrupts the corresponding party, it provides $\mathcal{A}$ with its internal state, and follows $\mathcal{A}$'s instructions. Additionally, if the dealer was corrupted, it will deliver the secret S along with the setup message $(\mathtt{Setup}, (t_1, \ldots, t_s), \mathtt{b}^{(1)}, \mathtt{b}^{(2)} \oplus S)$ for $(\mathtt{b}^{(1)}, \mathtt{b}^{(2)}) := G_{k,|S|}(\mathtt{b})$, randomly chosen $t_1, \ldots, t_s$ in $\{0,1\}^m$ and $\mathtt{b} \in \{0,1\}^k$.

The proof is completed by showing that $\mathcal{Z}$ cannot use the differences between the two runs to distinguish one from the other. And again, there is a single difference from the environment's perspective. It occurs in the case that no party is corrupted before setup. In the ideal-process, $\mathcal{Z}$ receives from $\mathcal{S}$ the values $(t'_1, \ldots, t'_s), c_1, c_2$ while it receives $(t_1, \ldots, t_s), \mathtt{b}^{(1)}, \mathtt{b}^{(2)} \oplus S$ from $\mathcal{A}$. The values $\mathtt{b}^{(1)}$ and $\mathtt{b}^{(2)}$ generated by D in the real-life protocol are computationally independent, whereas the values c_1 and $c_2 \oplus S$ generated by $\mathcal{S}$ in the ideal process are independently distributed. Since the values $t_1, \ldots, t_s$ and $t'_1, \ldots, t'_s$ are both drawn independently in $\{0,1\}^m$, it follows immediately that an environment that distinguishes one view from the other with non-negligible probability, breaks the security of $G_{k,|S|}$, which proves the theorem. □

A word on sampling. It is important to notice that the sampling in the ideal process and the selection process for $t_1, \ldots, t_s$ and $\mathtt{b}$ in the real-life protocol define identical distributions supported in K.

Remark 4.4 *The selection process for K in the real-life protocol is not straight forward. One chooses $b_1, \ldots, b_k$ randomly (same as in the ideal process) and then chooses $t_1, \ldots, t_s$ from which we the values $i_1, \ldots, i_k$ are deduced along with the information necessary to evaluate the auxiliary function σ. However, we could actually choose the value of the key $K = \{(i_1, b_1), \ldots, (i_k, b_k)\}$ and then deduce some values $t_1, \ldots, t_s \in \{0,1\}^m$ so that the function $\sigma_{(t_1,\ldots,t_s)}$ verifies $\sigma_{(t_1,\ldots,t_s)}(i_1, x) = \mathtt{b}$ if and only if $p_K(x) = \mathtt{true}$ (Property 1). Since the function $\sigma_{(t_1,\ldots,t_s)}(i_1, \cdot)$ is onto, for every $t_1, \ldots, t_s \in \{0,1\}^m$, we would like to argue that by changing the original setup method for the real-life protocol $\pi_{\mathtt{subset-s}}$ by this new setup method does not damage its security. However, it is unclear if one can actually do it.*

4.3 Multiple-strings trigger

Let $k, s \in \mathbb{Z}$ be integers with $s \geq 2$, where k is the security parameter. The ideal functionality for a multiple-strings trigger, $\mathcal{F}_{\texttt{mult}}$, is defined by the predicate family

$$\left\{p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}} : \{0,1\}^* \to \{\texttt{true},\texttt{false}\}; \mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)} \in \{0,1\}^k\right\},$$

where $p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}}$ is defined by $p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}}(x) := \texttt{true}$ if, writing $x = (x_1,\ldots,x_t)$, there exist indices $i_1,\ldots,i_s$ such that $(x_{i_1},\ldots,x_{i_1+k-1}) = \mathtt{b}^{(1)},\ldots,(x_{i_s},\ldots,x_{i_s+k-1}) = \mathtt{b}^{(s)}$, and $p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}}(x) := \texttt{false}$ if not. The sampling algorithm draws the values of $\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)} \in \{0,1\}^k$ according to sk bits of its random tape. Notice that the ideal functionality can feasibly evaluate predicates; it runs in $(sk)^{O(1)}$.

Real-life protocol ($\pi_{\texttt{mult-a}}$).– The dealer waits for an input of the form (Setup, S) and terminates if $|S| < k$ holds. Else, it draws strings $\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}$ in $\{0,1\}^k$ from it random tape, uses the pseudo-random generation algorithm $G_{k,|S|}$ to compute $(\mathtt{b}^{(1,1)},\mathtt{b}^{(1,2)}) := G_{k,|S|}(\mathtt{b}^{(1)}),\ldots,(\mathtt{b}^{(s,1)},\mathtt{b}^{(s,2)}) := G_{k,|S|}(\mathtt{b}^{(s)})$ in $\{0,1\}^k \times \{0,1\}^s$, and deletes the values of $\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}$ from its memory. Finally, it writes the message $(\mathtt{Setup}, \mathtt{b}^{(1,1)},\ldots,\mathtt{b}^{(s,1)}, S \oplus (\oplus_i \mathtt{b}^{(i,2)})$ in its outgoing communication tape with recipient T and terminates[6]. The triggerer expects a message of the form $(\mathsf{Setup}, A_1,\ldots,A_s,B)$, with $A_1,\ldots,A_s,B \in \{0,1\}^*$, and outputs $\mathsf{TriggerActivated}$. Then, for every input $(\mathtt{Check}, x)$ it receives, the triggerer outputs every substring $(x_i,\ldots,x_{i+k-1})$ such that the first entry in $G_{k,|S|}(x_i,\ldots,x_{i+k-1})$ equals A_j, for some i,j with $1 \leq i \leq |x|-k+1, 1 \leq j \leq s$. Additionally, it this condition holds for all $A_1,\ldots,A_s$, then the triggerer also outputs $B \oplus \big(\oplus_i (x_i,\ldots,x_{i+k-1})\big)$ —here the X-OR is taken over one substring $(x_i,\ldots,x_{i+k-1})$ matching A_j, for each j with $1 \leq j \leq k$ (without repetitions). Else, it returns nothing.

The real-life protocol $\pi_{\texttt{mult-a}}$ realizes the ideal functionality $\mathcal{F}_{\texttt{mult}}$ with respect to static adversaries. However, this ideal functionality does not mimic an "idealized computation" as expected, since the protocol $\pi_{\texttt{mult-a}}$ leaks some information (with negligible probability). Explicitly, if an input $x \in \{0,1\}^*$ is such that several, but not all, of the $\mathtt{b}^{(i)}$s are contained in x, then T discovers these values. Therefore, we need to modify the secure trigger ideal functionality replacing step (2) in Figure 1 by (2') as in Figure 4.

[6]The selection of the secret sharing scheme used to reconstruct the key $\oplus_i \mathtt{b}_i$ from the shares $\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}$ is arbitrary and could be modularly replaced with the same security results. We use this one here for the sake of simplicity.

Functionality $\mathcal{F}'_{\mathtt{mult}}$

(2') Wait for one of the messages $(\mathsf{Trigger}, \mathcal{F}'_{\mathtt{mult}})$ or $(\mathsf{Trigger}, \mathcal{S})$ from $\mathcal{S}$.

- a) If the message was $(\mathsf{Trigger}, \mathcal{F}'_{\mathtt{mult}})$, for every input (Check, x) received from T, for every $i, j \in \mathbb{Z}$, with $1 \leq i < |x| - k, 1 \leq j \leq s$ such that $(x_i, \ldots, x_{i+k-1}) = \mathtt{b}^{(j)}$, output $(x_i, \ldots, x_{i+k-1})$ for T. (Here $|x|$ denotes the size of the bitstring x.) Additionally, if $p_{\mathtt{b}^{(1)},\ldots,\mathtt{b}^{(s)}}(x) = \mathtt{true}$ holds, return the secret to T.

- b) If it was $(\mathsf{Trigger}, \mathcal{S})$, for any input (Check, x) received from T, hand it to $\mathcal{S}$. If $\mathcal{S}$ answers this message, forward the answer to T.

Figure 4: Modified Ideal Functionality

Theorem 4.5 *Protocol $\pi_{\mathtt{mult-a}}$ securely realizes $\mathcal{F}'_{\mathtt{mult}}$ with respect to adaptive adversaries if local erasure of data is allowed.*

Proof: The proof follows the lines of the proofs of Theorems 4.2 and 4.3. Given a real-life adversary, we design an ideal-process adversary such that no environment can distinguish one from the other with non-negligible probability. Again, the ideal-process adversary, $\mathcal{S}$, will simulate a real-life adversary, $\mathcal{A}$, imitate a copy of $\pi_{\mathtt{mult-a}}$ for $\mathcal{A}$, and act as an interface between $\mathcal{A}$ and $\mathcal{Z}$.

Assume that the dealer is not corrupted before setup. When the ideal-process adversary receives the size of the secret from $\mathcal{F}_{\mathtt{mult}}$, it sends the setup message $(\mathsf{Setup}, c_1, \ldots, c_s, \oplus_i d_i)$ to $\mathcal{A}$, where $c_1, \ldots, c_s$ and $d_1, \ldots, d_s$ are uniformly selected in $\{0,1\}^k$ and $\{0,1\}^{|S|}$, respectively. Next, the ideal-process adversary waits for $\mathcal{A}$ to forward a setup message to the triggerer, and starts letting through messages from $\mathcal{F}'_{\mathtt{mult}}$ to T. If the real-life adversary forwarded the setup message without modifications, then $\mathcal{S}$ sends $(\mathsf{Trigger}, \mathcal{F}'_{\mathtt{mult}})$ to the ideal functionality. If the setup message was modified to $(\mathsf{Setup}, A_1, \ldots, A_s, B)$ (or created), for $A_1, \ldots, A_s, B \in \{0,1\}^*$, the ideal-process adversary sends the message $(\mathsf{Trigger}, \mathcal{S})$ to the ideal functionality. Next, for every input (Check, x) it receives from $\mathcal{F}'_{\mathtt{mult}}$, for every i, j, with $1 \leq i \leq |x|-k+1, 1 \leq j \leq s$, such that the first entry of $G_{k,|S|}(x_i, \ldots, x_{i+k-1})$ agrees with A_j, the ideal-process adversary outputs $(x_i, \ldots, x_{i+k-1})$. Also, if all the $A_1, \ldots, A_s$ were matched, $\mathcal{S}$ hands $\left(\oplus_j (x_{i_j}, \ldots, x_{i_j+k-1})\right) \oplus B)$ to $\mathcal{F}'_{\mathtt{mult}}$.

In case that $\mathcal{A}$ corrupts D or T, then $\mathcal{S}$ corrupts the corresponding party and provides $\mathcal{A}$ with its internal state. If D is corrupted before $|S|$ has been received, then $\mathcal{S}$ will simulate a "real" dealer for $\mathcal{A}$: it will compute the setup message $(\mathsf{Setup}, \mathtt{b}^{(1,1)}, \ldots, \mathtt{b}^{(1,s)}, (\oplus_i \mathtt{b}^{(2,i)}) \oplus S)$, for $(\mathtt{b}^{(1,i)}, \mathtt{b}^{(2,i)}) := G_{k,|S|}(\mathtt{b}^{(i)})$ and $\mathtt{b}^{(i)} \in \{0,1\}^k$ uniformly chosen (for all i with $1 \leq i \leq s$), and send it to T along with the secret S. If D is corrupted after setup, (thus, the values of $\mathtt{b}^{(1)}, \ldots, \mathtt{b}^{(s)}$ have already been deleted in the real-life protocol), then the ideal-process adversary obtains the secret S from the dealer and hands $\mathcal{A}$ the strings $(c_1, d_1), \ldots, (c_{s-1}, d_{s-1}), (c_s, d_s \oplus S)$ representing values that are consistent with the setup message and secret.

In order to show that protocol $\pi_{\mathtt{mult-a}}$ securely realizes $\mathcal{F}'_{\mathtt{mult}}$, we notice that there is a single difference between the two runs. The values $(\mathtt{b}^{(1,1)}, \mathtt{b}^{(2,1)}), \ldots, (\mathtt{b}^{(1,s)},$
$\mathtt{b}^{(2,s)})$ generated by D in the real-life protocol are computationally independent, whereas the values $(c_1, d_1), \ldots, (c_{s-1}, d_{s-1}), (c_s, d_s \oplus S)$ generated by $\mathcal{S}$ in the ideal process are independently distributed. However, one can see that breaking the security of this protocol can be reduced to breaking the security of $G_{k,|S|}$, which proves the theorem. □

Finally, it is interesting to compare the subsequence trigger with the multiple-strings trigger. A first comparison shows that $\mathcal{F}_{\mathtt{subset}}$ accepts fixed-size inputs while $\mathcal{F}'_{\mathtt{mult}}$ accepts arbitrary inputs. Even more, they provide different security, for example, assume that we want to trigger a secret procedure if an input $x = (x_1, \ldots, x_{1024})$ verifies the conditions $(x_1, \ldots, x_{10}) = (k_1, \ldots, k_{10})$, $(x_{101}, \ldots, x_{110}) = (k_{11}, \ldots, k_{20})$, $\ldots, (x_{700}, \ldots, x_{710}) = (k_{71}, \ldots, k_{80})$, for some secret values $(k_1, \ldots, k_{80}) \in \{0,1\}^{80}$. That is, we want to check if 8 different strings within the input match 8 secret strings, these strings being secret. If we use the multiple-strings trigger then an attacker can succeed by doing a brute-force attack on 8 strings of 10 bits, each. In fact, by using the subsequence trigger ensure that attackers do not learn which are the bits being checked. However, if we use a subsequence trigger, then we can get 80 bits security!

5 Applications

We return to the example "anonymous shopping agent" from Section 1.1. In this setting, etailers provide an anonymous shopping service whereby customers are allowed to submit an agent that crawls databases searching for the product they like. A customer wants to use an agent to crawl these databases for his targeted product without the product's description nor his

name. It is required that the code resists reverse-engineering analysis and does not leak the sensitive information.

To design the mobile code we will make use of the subsequence trigger. But first, let us state our assumptions.

- Let us assume without loss of generality that the etailer databases have all the same format (or support a single crawling API), and each entry can be described by a bitstring of size N, for some $N \in \mathbb{Z}$.
- Each entry is described by a set of attributes of a fixed length,
- Customers can deploy their mobile code anonymously,
- Etailers can run the agents safely (e.g., sandboxing),
- The number of possible products on etailers databases is large (e.g., of size larger than 2^{80}). Moreover, we can assume that the distribution deduced from marketing information (and all that other information available to etailers) is such that the number of products with probability greater than the target product's own probability is large, too.

To design the trigger, we write the description of the selected product and its price as a bitstring of a considerable size (e.g., $k \geq 160$). To do this, we uniformly choose certain bits from the target product description, writing their index and their value as $K \subset \{1, \ldots, N\} \times \{0, 1\}$ (and possibly add some pairs (i, b) so that the product, and its preferred price, are described unequivocally by K). Finally, we run the setup for a subsequence trigger real-life protocol and compute a triggerer T with K selected as above and S to be a procedure that sends an email to the buyer with a description of the product it found and the name of the etailer.

From the above assumptions one can deduce that no brute-force attack on this secure trigger can succeed, and therefore the agent is secure.

References

[Ano02] Anonymous. Slashdot post (#4537102). At `http://www.slashdot.org`, 2002.

[BFN+03] Diego Bendersky, Ariel Futoransky, Luciano Notarfrancesco, Carlos Sarraute, and Ariel Waissbein. Advanced software protection now. Corelabs Technical Report, available at `http://www.coresecurity.com/corelabs/projects/software_protection.php`, 2003.

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18, UCSB, Santa Barbara (CA), August 2001. Springer.

[Blu81] Manuel Blum. Coin flipping by telephone. In Allen Gersho, editor, *Advances in Cryptology. A report on CRYPTO '81, IEEE Workshop on Communications Securit*, pages 11–15, Santa Barbara, California, USA, August 1981.

[BN00] Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, *AAdvances in Cryptology - CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254, Santa Barbara, California, USA, 2000. Springer.

[Can97] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97*, volume 1294 of *LNCS*, pages 455–469, Santa Barbara, California, USA, 1997. Springer.

[Can00a] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[Can00b] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information (revised version), 2000.

[Can00c] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067. Full paper version of [Can01], 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, Proceedings,*, pages 136–145, Las Vegas, Nevada, USA, 14th-17th October 2001, 2001. IEEE Computer Society.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40, Santa Barbara, Ca) USA, 2001. Springer.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 337–351, Amsterdam, The Netherlands, 2002. Springer.

[CMR98] Ran Canetti, Daniele Micciancio, and Omer Reingold. Perfectly one-way probabilistic hash functions (preliminary version). In *Thirtieth Annual ACM Symposium on the Theory of Computing. Proceedings,*, pages 131–140, Dallas, Texas, May 1998. ACM Press.

[CPV03] Joris Classens, Bart Preneel, and Joss Vandewalle. (How) can mobile agents do secure electronic transactions on untrusted hosts? — a survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3(1361):28–48, 2003.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92*, volume 740 of *LNCS*, pages 139–147, UCSB, Santa Barbara (CA), USA, August 1993. Springer.

[DN02] Ivan Damgaard and Jesper Buus Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *LNCS*, pages 581–596, Santa Barbara, California, USA, 2002. Springer.

[DOR99] Giovanni Di Crescenzo, Rafail Ostrovsky, and Sivaramakrishnan Rajagopalan. Conditional oblivious transfer and timed-release encryption. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *Add data for field: Series*, pages 74–89, Prague, Czech Republic, May 1999. Springer.

[Gol01] Oded Goldreich. *Foundations of Cryptography (Vol. 1).* Cambridge University Press, 2001.

[Gol04] Oded Goldreich. *Foundations of Cryptography (Vol. 2).* Cambridge University Press, 2004.

[Gry92] Dmitry Gryaznov. An analysis of cheeba. In *EICAR'92 conference*, 1992.

[Hoh98] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 92–113. Springer, 1998.

[JS02] A. Juels and M. Sudan. A fuzzy vault scheme. In *Proceedings of IEEE Internation Symposium on Information Theory*, pages 408–426, Lausanne, Switzerland, 2002. IEEE Press.

[LPS04] Benjamin Lynn, Manoj Prahbakasan, and Amit Sahai. Positive results and techniques for obfuscation. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology — Eurocrypt '04*, volume 3027 of *LNCS*, pages 20–39, Interlaken, Switzerland, May 2004. Springer-Verlag, New York.

[Per03] Frederic Perriot. Personal communication, 2003.

[Riv97] Ronald L. Rivest. All-or-nothing encryption and the package transform. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97*, volume 1267 of *LNCS*, pages 210–218, Haifa, Israel, 1997. Springer.

[RSW96] Ron Rivest, Adi Shamir, and David Wagner. Time lock puzzles and timed release cryptography. Technical report, MIT Laboratory of Computer Science, 1996.

[vO03] Paul C. van Oorschot. Revisiting software protection (invited talk). In Colin Boyd and Wenbo Mao, editors, *Information Security, 6th International Conference, ISC 2003*, volume 2851 of *LNCS*, pages 1–13, Bristol, UK, October 2003. Springer.