Safe enclosures: towards cryptographic techniques for server protection

Sergiu Bursuc[1] and Julian P. Murphy[2]

[1] School of Computer Science, University of Bristol
[2] Centre for Secure Information Technologies, Queen's University of Belfast

Abstract. Cryptography is generally used to protect sensitive data from an untrusted server. In this paper, we investigate the converse question: can we use cryptography to protect a trusted server from untrusted data? As a first step in this direction, we propose the notion of safe enclosures. Intuitively, a safe enclosure is a cryptographic primitive that encapsulates data in a way that allows to perform some computation on it, while at the same time protecting the server from malicious data. Furthermore, a safe enclosure should come equipped with a dedicated protocol that implements the enclosing function with unconditional integrity. Otherwise, unguarded data may reach the server. We discuss the novelty of these concepts, propose their formal definition and show several realizations.

1 Introduction

There is a pattern that can be generally noticed when security breaches arise: data on a small device, usually belonging to an employee of a company, is first corrupted and serves as a ladder step for malware to penetrate into a bigger system. Such infection is difficult to prevent, especially when it is based on zero-day vulnerabilities and hence invisible to any antivirus. Obviously, encloistering the server with a barrier that does not let anything in would work, but it is not a realistic option.

Based on the insight of the ongoing work on fully homomorphic encryption [1–3], we ask the question: can computation on "encrypted" data serve as a paradigm for protecting the server from data that is possibly infected by malware? The purpose of this paper is to propose initial ideas and set the stage for further research in this area. Crucially, we are looking for an appropriate crypographic notion, and encryption may be only an instance of it.

We are looking for a way to cryptographically enclose a message inside a security perimeter. The enclosure can be transparent and may not hide the message at all. At the same time, it is important that enclosed data can not go outside the cryptographic perimeter and leak unguarded into the server. Furthermore, an attacker who can choose the message to be enclosed, shoud not be able to influence any part of the resulting enclosure. Otherwise, the security perimeter itself could be used to carry malicious content. In section 3 we provide further motivation and definitions for the desired cryptographic primitive. In section 4

we discuss why current cryptographic notions, although related, are inadequate for the considered problem.

One particular challenge that arises from the definition of a safe enclosure is the need for an enclosing protocol that functions with unconditional integrity (in fact, a weaker notion of *unconditional enclosure* will be sufficient). Indeed, we can not rely on untrusted clients to enclose the data before it is being uploaded to the server. Neither can we upload the data first and then safely enclose it, because the server may be compromised by that time. In section 5, we propose a generic solution for unconditional enclosure based on two notions:

- *homomorphic recomposability*, which means that small basic enclosed messages can be combined into the enclosure of a bigger message.
- *enclosing transducer*, which is a program that can receive as input only basic requests and can output *only* basic enclosed messages.

In section 6, we present the design of an enclosing transducer based on trusted computing: a trusted CPU, a trusted TPM, dynamic measurement of programs and sealed storage. Putting all these elements together, we instantiate our framework in section 7. In particular, we use homomorphic encryption to implement an enclosure function. As we argue in conclusion, this is only an instance, which may offer an unnecessarily high level of protection. In general, our paper opens new questions both about the appropriate security definitions and about the appropriate cryptographic primitives for server protection.

2 Preliminaries

We briefly present some high-level aspects of encryption schemes and of trusted computing, that are sufficient for understanding the paper.

Encryption schemes. A public key encryption scheme is defined by a triple $(\mathcal{K}, \mathcal{E}, \mathcal{D})$, where $\mathcal{K}$ is a key generation algorithm, $\mathcal{E}$ is an encryption algorithm, and $\mathcal{D}$ is a decryption algorithm. We write $(pk, sk) \in \mathcal{K}$ if (pk, sk) is a public/private key pair returned by $\mathcal{K}$. Then, for all message m, we have $\mathcal{D}(\mathcal{E}(m, pk), sk) = m$. $\mathcal{E}(m, pk)$ will be called a ciphertext and m is the corresponding plaintext. To keep the notation simple, we leave implicit the random element of encryption schemes, but use the notation $\mathcal{E}(m, pk, r)$ when the random r is important.

For an algorithm $\mathcal{A}$, we denote by $\mathcal{A}[m]$ the result of applying $\mathcal{A}$ on input m.

Trusted computing. A trusted platform module (TPM) is a piece of hardware that offers some protected computational capabilities. It can store the *measurement* of loaded programs and provide evidence about the state of the platform. Private information can be *sealed* against the measurment of a specific program P, and can be unsealed by the TPM only when its registers attest that P is loaded. In this way, no matter what is the initial software configuration, a

trusted program can be loaded and given access to private data in a protected environment [4, 5].

We denote the measurement of a program P by $h(P)$. For instance, the measurement of P can be obtained by applying a hash function to the code of P. We denote the sealing of a value v against a measurement m by $\mathcal{S}(v, m)$. Then, if the measurement of a program P is equal to m, P can have access to v in an environment protected by trusted hardware.

3 Safe enclosures: motivation and definition

In terms of functionality, an enclosure scheme looks like a homomorphic encryption scheme, without keys:

Definition 1 (enclosure scheme). *An* enclosure scheme *is given by a tuple* $(\mathcal{E}, \mathcal{D}, \mathcal{H})$*, where*

- $\mathcal{E}$ *is an* enclosing function*: a function that associates a message* $\mathcal{E}(m)$ *to any input message* m
- $\mathcal{D}$ *is a* disclosing function*: for all message* m*,* $\mathcal{D}(\mathcal{E}(m)) = m$*.*
- $\mathcal{H}$ *is a set of pairs of functions that specifies* computation on enclosed data*: for all* $(\mathcal{F}_1, \mathcal{F}_2) \in \mathcal{H}$ *and all list of enclosed messages* $\mathcal{E}(m_1), \ldots, \mathcal{E}(m_n)$*, we have:*

$$\mathcal{F}_1(\mathcal{E}(m_1), \ldots, \mathcal{E}(m_n)) = \mathcal{E}(\mathcal{F}_2(m_1, \ldots, m_n))$$

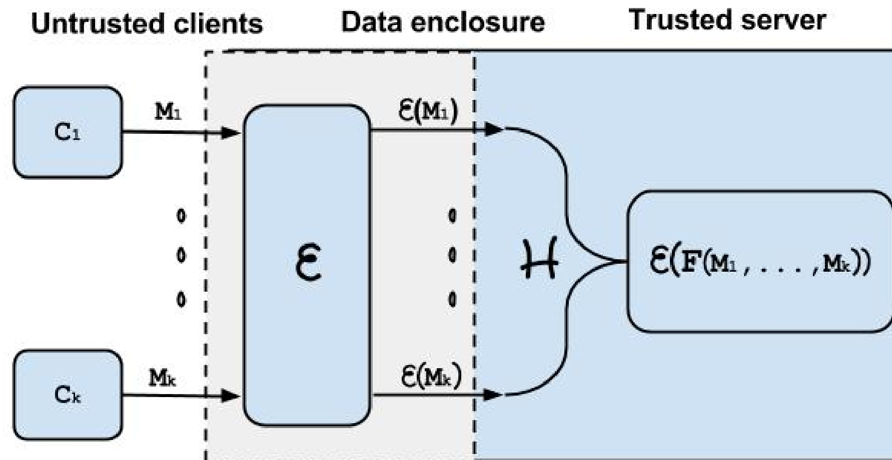

Fig. 1. Server protection by data enclosure

In the context of the untrusted clients / trusted server scenario that we discussed in introduction, the use of an enclosure scheme is depicted in figure 1. The

idea is that data from clients is enclosed under the function $\mathcal{E}$ and computation on the server can be performed relying on the homomorphic properties $\mathcal{H}$.

In terms of security, a safe enclosure should ensure that enclosed data can not affect the server. This requirement is modeled in definition 2 by a relation on messages and is more abstract than the security requirement for an encryption scheme. Furthermore, we require a protocol that securely implements the enclosure function, such that data can not reach unenclosed on the server. This second requirement is more concrete and operational, but it is equally important for the applicability of a safe enclosure:

Definition 2 (safe enclosure). *We say that an enclosing scheme $(\mathcal{E}, \mathcal{D}, \mathcal{H})$ is a* safe enclosure *with respect to a triple $(\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E}, \sim_\mathcal{H})$, if the following hold:*

- $\sim_\mathcal{H}$ *is a relation among messages that describes* safety of enclosed messages*: for all messages $\mathcal{E}(m_1)$ and $\mathcal{E}(m_2)$, we have*

$$\mathcal{E}(m_1) \sim_\mathcal{H} \mathcal{E}(m_2)$$

- $\mathcal{U}_\mathcal{E}$ *and* $\mathcal{T}_\mathcal{E}$ *are algorithms that describe an* enclosure protocol*: for all message m, we have*

$$\mathcal{T}_\mathcal{E}[\, \mathcal{U}_\mathcal{E}[\, m \,] \,] = \mathcal{E}(m)$$

- unconditional enclosure*: for all algorithm $\mathcal{U}'_\mathcal{E}$ and message m, we have*

$$\mathcal{T}_\mathcal{E}[\, \mathcal{U}'_\mathcal{E}[\, m \,] \,] = \mathcal{E}(m')$$

  for some message m'.

Intuitively, the relation $\sim_\mathcal{H}$ should be defined such that, even when a message m_1 carries malicious content, after being enclosed as $\mathcal{E}(m_1)$ it is "indistinguishable" from any other (honest) message m_2, being enclosed as $\mathcal{E}(m_2)$. This notion of indistinguishability may depend on the computation on enclosed messages that is performed on the server, and that is why it may be parametrized by $\mathcal{H}$.

Let us now explain the enclosing protocol $(\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E})$, whose goal is to produce $\mathcal{E}(m)$ from m. The role of $\mathcal{U}_\mathcal{E}$ is to specify any preprocessing of the message m that can be performed in an *untrusted environment*. Even if the environment where $\mathcal{U}_\mathcal{E}$ is executed is compromised, the point 3 of the definition ensures data provided to the server will still be enclosed, although it may be compromised (i.e. the message m' may be different from m). On the other hand, the algorithm $\mathcal{T}_\mathcal{E}$ is to be performed in a trusted environment on the server or on a proxy. Crucially, the definition assumes that the $\mathcal{T}_\mathcal{E}$ can not be compromised in its interaction with a malicious $\mathcal{U}_\mathcal{E}$. This is a property that should be ensured by careful design of $\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E}$ and of their interaction. We present a particular design in sections 5 and 6.

What security properties are not covered by safe enclosures. As discussed above, it is out of our scope to ensure that data under the enclosure is not compromised. Even more, it could be the case that some data from an honest client may be compromised after it reaches the server. Indeed, enclosed honest

data $\mathcal{E}(m_h)$ may be homomorphically combined with enclosed dishonest data $\mathcal{E}(m_d)$, and there is no guarantee about the result $\mathcal{E}(m_r)$. For instance, m_d may trigger a malicious instruction to delete all records from an enclosed database. On the other hand, a safe enclosure does guarantee that data on the server that is not enclosed (or is enclosed with a different function $\mathcal{E}'$) can not be compromised by malicious clients. In particular, malicious clients can not compromise the software that runs the server.

4 Limitations of current methods

In the spectrum of current solutions for data security, there are two concepts that one may relate to safe enclosures: verifiable computation and hardware-based security. In this section we discuss why these notions alone are not enough to implement safe enclosures. Mainly, the difficulty lies in the property of *unconditional enclosure* from definition 2: there should be a trusted algorithm $\mathcal{T}_\mathcal{E}$ that ensures all its outputs are enclosed, no matter its execution environment.

4.1 The computation-verification dissociation

Verifiable computation, e.g. [6–9], allows a trusted party to check that a computation has been performed correctly. Then, given an enclosing scheme $(\mathcal{E}, \mathcal{D}, \mathcal{H})$, we could:

- devise a verification protocol $\mathcal{V}_\mathcal{E}$ such that $\mathcal{V}_\mathcal{E}(m, \pi) = 1$ if and only if π is a valid proof of the fact that m represents an enclosed message, i.e. $m = \mathcal{E}(m')$, for some message m'.
- outsource the task to enclose incoming messages to a proxy, that is not necessarily trusted.
- upon receiving messages from the proxy, verify that they are properly enclosed, by relying on $\mathcal{V}$.

However, in this scenario, the verification of the computation comes too late: data from untrusted clients could compromize the proxy, which in turn could compromize the server by providing a malicious proof. Formally, in order to satisfy the requirement of unconditional enclosure, there should be a way to confine the proof verification to a trusted execution environment that can not be compromized. In fact, this brings us back to where we started.

4.2 The hardware-security association

Hardware-based security, e.g. [5, 10–13], could allow a form of unconditional enclosure by relying on trusted hardware to perform secure computation, independent of the software environment. The fact that secure processing is tightly related to hardware opens two options:

- either the trusted hardware is specific for the computation that needs to be performed, e.g. [13, 14]

– or else the hardware is generic, and it only allows to attest that some software is running in a secure environment, e.g. [5, 10–12]

Specific hardware would work, but it is not realistic to assume a dedicated processor for various enclosing functions. The second option - a mix of trusted software and trusted hardware - seems more practical. However, the software-hardware interaction, and the software itself, must be carefully designed to avoid attacks. Our paper aims to minimize the amount of software that has to be trusted in this context. Indeed, we would like to avoid the following trivial, but dangerous, solution: simply take a program implementing an enclosing function $\mathcal{E}$ and execute it on a trusted platform. This solution is no better than others, because it still assumes that the program $\mathcal{E}$ can not be compromised after its interaction with an untrusted input. That is why, in section 5 and 6, we aim for a solution where the trusted software is minimalistic, and it does not process any untrusted inputs.

5 A generic enclosure protocol

In this section we present a particular enclosure protocol that achieves unconditional enclosure, relying on a specific property of the enclosing function:

Definition 3 (homomorphic recomposability). *We say that an enclosing function $\mathcal{E}$ satisfies* homomorphic recomposability *if there is a set of messages $a_1, \ldots, a_k$ and two algorithms $\mathcal{A}_{\mathsf{dec}}, \mathcal{A}_{\mathsf{rec}}$ such that, for all message m, there are $\{m_1, \ldots, m_n\} \subseteq \{a_1, \ldots, a_k\}$ such that*

$$\begin{aligned} \mathcal{A}_{\mathsf{dec}}(m) &= (m_1, \ldots, m_n) \\ \mathcal{A}_{\mathsf{rec}}(\mathcal{E}(m_1), \ldots, \mathcal{E}(m_n)) &= \mathcal{E}(m) \end{aligned}$$

For any $m \in \{a_1, \ldots, a_k\}$, we let $\langle m \rangle = i$ if $m = a_i$. The messages $a_1, \ldots, a_k$ will also be called atomic messages. Thus, homomorphic recomposability states that any message can be decomposed in atomic messages, relying on $\mathcal{A}_{\mathsf{dec}}$, and that its enclosure can be obtained by combining the enclosures of its atomic components, relying on $\mathcal{A}_{\mathsf{rec}}$.

Next, we give the specification of a program that can only output enclosed atomic messages, and nothing else:

Definition 4 (enclosing transducer). *We say that a program $\mathcal{A}_{\mathcal{E}}$ is an* enclosing transducer *for messages $a_1, \ldots, a_k$ and enclosing function $\mathcal{E}$ if for all $i \in \{1, \ldots, k\}$, we have $\mathcal{A}_{\mathcal{E}}[i] = \mathcal{E}(a_i)$.*

Furthermore, we say that $\mathcal{A}_{\mathcal{E}}$ is a safe enclosing transducer *if for any input i', we have $\mathcal{A}_{\mathcal{E}}[i'] = \mathcal{E}(m')$, for some message m'.*

Intuitively, given atomic messages $a_1, \ldots, a_k$, an enclosing transducer simply produces $\mathcal{E}(a)$ from $\langle a \rangle$. Additionaly, the safety requirement ensures that no matter what input is given to the transducer (possibly outside the set $\langle a_1 \rangle, \ldots, \langle a_k \rangle$), its output is still an enclosed message.

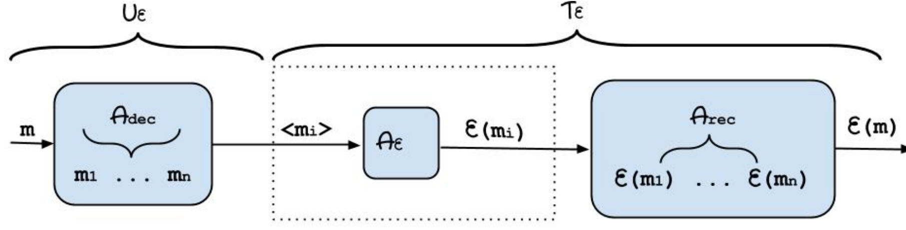

Fig. 2. Safe enclosure based on homomorphic recomposability and atomic transducer

Enclosure protocol. According to definition 2, for an enclosure protocol we have to provide an algorithm $\mathcal{U}_\mathcal{E}$ to be executed in an untrusted environment and an algorithm $\mathcal{T}_\mathcal{E}$ to be executed in a trusted environment. As illustrated in figure 2, the algorithm $\mathcal{U}_\mathcal{E}$ consists in decomposing the input message m into a list of atomic messages $(m_1, \ldots, m_n)$, according to $\mathcal{A}_{\mathsf{dec}}$. For every m_i, its corresponding index $\langle m_i \rangle$ is transmitted to $\mathcal{T}_\mathcal{E}$ for enclosure. The algorithm $\mathcal{T}_\mathcal{E}$ consists in passing all message indices through the enclosing transducer and recomposing its outputs in an enclosure $\mathcal{E}(m)$ of m, relying on the algorithm $\mathcal{A}_{\mathsf{rec}}$.

A safe enclosing transducer has to determine a way of securely receiving $\langle a_1 \rangle, \ldots, \langle a_n \rangle$, of computing and of returning $\mathcal{E}(a_1), \ldots, \mathcal{E}(a_k)$. The fundamental observation here is that we have reduced the general problem of securely computing $\mathcal{E}(m)$, for *any message* m, to the more specific problem of computing $\mathcal{E}(a_1), \ldots, \mathcal{E}(a_k)$, for some *fixed messages* $a_1, \ldots, a_k$. This reduction is formalized in proposition 1:

Proposition 1. *Let* $(\mathcal{E}, \mathcal{D}, \mathcal{H})$ *be an enclosure scheme such that* $\mathcal{E}$ *satisfies homomorphic recomposability with respect to* $a_1, \ldots, a_k, \mathcal{A}_{\mathsf{dec}}, \mathcal{A}_{\mathsf{rec}}$*. Let* $\mathcal{A}_\mathcal{E}$ *be an enclosing transuducer and* $(\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E})$ *be an enclosure protocol built from* $\mathcal{E}$ *and* $\mathcal{A}_\mathcal{E}$ *as explained in figure 2. If:*

- $\mathcal{A}_\mathcal{E}$ *is a safe enclosing transducer and*
- *for all messages* m_1 *and* m_2*, we have* $\mathcal{E}(m_1) \sim_\mathcal{H} \mathcal{E}(m_2)$

then $(\mathcal{E}, \mathcal{D}, \mathcal{H})$ *is a safe enclosure scheme with respect to* $(\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E}, \sim_\mathcal{H})$*.*

Proof sketch: The first two points of the definition 2 follow immediately from the definition 4 and from the assumptions of the proposition. We are left to show unconditional enclosure. Assume by contradiction that there is $\mathcal{U}'_\mathcal{E}$ and a message m such that $\mathcal{T}_\mathcal{E}[\ \mathcal{U}'_\mathcal{E}[\ m\ ]\ ] \neq \mathcal{E}(m')$, for all message m'. From the properties of $\mathcal{A}_{\mathsf{rec}}$ and from the design of $\mathcal{T}_\mathcal{E}$, it follows then that there is an output m_0 of $\mathcal{A}_\mathcal{E}$ such that $m_0 \neq \mathcal{E}(m')$, for all message m'. This contradicts the definition of a safe enclosing transducer, and we can conclude. $\square$

Now, to complete the specification of $\mathcal{T}_\mathcal{E}$, we have to design a safe enclosing transducer. We propose a design based on trusted computing in the next section.

6 A trusted enclosing transducer

In this section we consider the problem of designing a safe enclosing transducer, according to definition 4. In fact, we can frame the problem in a slightly more general setting, without relation to an enclosing function: given a list of messages $\mathcal{E}(a_1), \ldots, \mathcal{E}(a_k)$, we have to design a program that can only output $\mathcal{E}(a_1), \ldots, \mathcal{E}(a_k)$, in any execution environment. In a normal execution, on input i, the program has to output $\mathcal{E}(a_i)$.

Private channel abstraction. We assume that the ability to output messages on behalf of the system on a channel c is restricted to agents that have knowledge of some private token tc. In a Dolev-Yao model (e.g. [15]), this amounts to a private channel, whose implementation we assume to be secure [16].

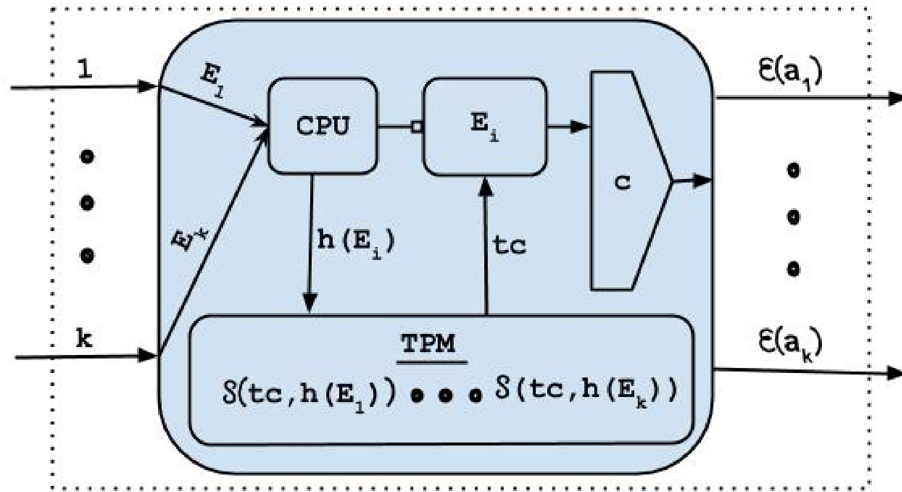

Fig. 3. Enclosing transducer based on trusted computing

Transducer design. We assume given programs $\mathsf{E}_1, \ldots, \mathsf{E}_k$ such that, for all $1 \leq i \leq k$, $\mathcal{E}_i$ outputs $\mathcal{E}(a_i)$ without taking any input from the environment. For instance, the value $\mathcal{E}_i(a)$ may be somehow hardcoded in the source code of the program E_i. The design of the enclosing transducer is illustrated in figure 3 and can be summarized as follows:

1. For every input i, the source code for the corresponding program E_i is passed to a trusted CPU. The process of transmitting E_i to the CPU does not have to be trusted and can be assumed to be in the control of the intruder. In particular, the source code claimed to be that of E_i may be compromised. Let us denote the actual program that is received by the CPU at this step

by $\mathsf{E'}_i$, which is equal to E_i in a honest execution, and may be different from E_i in a dishonest execution.
2. The value of the private token tc is sealed into the TPM against the measurement of each E_i, i.e. the values $\mathcal{S}(\mathsf{tc}, h(\mathsf{E}_1)), \ldots, \mathcal{S}(\mathsf{tc}, h(\mathsf{E}_k))$ are stored on the TPM. Crucially, these measurements correspond to the correct, uncorrupted source code of the programs $\mathsf{E}_1, \ldots, \mathsf{E}_k$.
3. The trusted CPU computes the measurement $h(\mathsf{E'}_i)$ of the program $\mathsf{E'}_i$ that is supplied by the untrusted software environment at the step 1, and communicates it to the TPM.
4. $\mathsf{E'}_i$ is executed by the CPU in a trusted environment and, if the measurement provided by the CPU attests that $\mathcal{E}_i$ has not been compromised, i.e. $\mathsf{E'}_i$ is equal to $\mathsf{E_i}$, the TPM can unseal the token tc for E_i, that can then forward its output on c.

It is important to note that the programs $\mathsf{E}_1, \ldots, \mathsf{E}_k$ do not accept any input from their execution environment, except the value tc that is received from the TPM on a trusted channel.

Let us compare this design with the simple option of executing an enclosure algorithm $\mathcal{E}$ in an environment protected by trusted hardware. The problem with the latter version is that we can not trust the input of the enclosure algorithm, and there is no way to anticipate it beforehand because the message space is unbounded. Thus, a malicious input combined with an implementation bug in the algorithm could lead to outputs that are not enclosed, breaking the unconditional enclosure requirement of definition 2. On the other hand, the algorithms $\mathsf{E}_1, \ldots, \mathsf{E}_m$ are fixed and predetermined beforehand, and they accept no inputs from outside. We can measure the "enclosure" program that is supplied as input to the trusted execution environment and we can be sure that only uncompromised programs have access to c. Indeed, we have the following proposition:

Proposition 2. *Assume that the following hold for the process $\mathcal{A}_\mathcal{E}$ described in figure 3:*

1. *the* TPM *and the* CPU *are honest*
2. *the programs $\mathsf{E}_1, \ldots, \mathsf{E}_k$ correctly compute $\mathcal{E}(a_1), \ldots, \mathcal{E}(a_k)$ and output the result on c. Furthermore, $\mathcal{E}(a_i)$ is the only output of E_i.*
3. *the only way to output data from $\mathcal{A}_\mathcal{E}$ to the execution environment is to have access to the token tc*
4. *the token tc is only used as described in the specification of $\mathcal{A}_\mathcal{E}$*

Then, $\mathcal{A}_\mathcal{E}$ is a safe enclosing transducer for $a_1, \ldots, a_k$ and $\mathcal{E}$.

Proof sketch: The fact that $\mathcal{A}_\mathcal{E}$ is an enclosing transducer follows immediately from its definition. In addition, we also have to show that it is a safe one. Assume by contradiction that $\mathcal{A}_\mathcal{E}$ is not safe, i.e. there is a message i' such that $\mathcal{A}_\mathcal{E}[i'] \neq \mathcal{E}(m')$, for all message m'. From asumption 3, it follows that $\mathcal{A}_\mathcal{E}[i']$ has been output on the channel c by a party who has access to tc. From the assumptions 1 and 4, it follows that the only processes that obtain tc are $\mathsf{E}_1, \ldots, \mathsf{E}_k$.

Furthermore, the second part of assumption 2 ensures that the token tc is kept secret by $\mathsf{E}_1, \ldots, \mathsf{E}_k$. This means that the only way we can have $\forall m'.\mathcal{A}_{\mathcal{E}}[i'] \neq \mathcal{E}(m')$ is an error in one of $\mathsf{E}_1, \ldots, \mathsf{E}_k$. This is again ruled out by assumption 2, and we obtain a contradiction. $\square$

7 Safe enclosures: an instance

In this section we instantiate our framework by providing an enclosure scheme $(\mathcal{E}, \mathcal{D}, \mathcal{H})$, based on homomorphic encryption, that is safe with respect to a triple $(\mathcal{U}_{\mathcal{E}}, \mathcal{T}_{\mathcal{E}}, \sim_{\mathcal{H}})$, where $\sim_{\mathcal{H}}$ is based on the IND-CPA security of encryption and $\mathcal{U}_{\mathcal{E}}, \mathcal{T}_{\mathcal{E}}$ are based on the generic constructions from sections 5 and 6.

Let $(\mathcal{K}, \mathcal{E}_0, \mathcal{D}_0)$ be a public-key encryption scheme and let us fix a key $(pk, sk) \in \mathcal{K}$ and a random number r. Now, for all message m, we define:

$$\begin{aligned} \mathcal{E}(m) &= \mathcal{E}_0(m, pk, r) \\ \mathcal{D}(m) &= \mathcal{D}_0(m, sk) \end{aligned}$$

Furthermore, if the scheme is homomorphic with respect to addition [17], multiplication [18, 19] or both [1], we gather in the set $\mathcal{H}$ the pairs of functions that implement the homomorphism. Concretely, in the case of fully-homomorphic encryption [1], we let $\mathcal{H} = \{(\star_{\mathcal{E}}, \star), (+_{\mathcal{E}}, +)\}$, where $\star$ and $+$ are multiplication and addition, while $\star_{\mathcal{E}}$ and $+_{\mathcal{E}}$ are the functions that implement multiplication and addition over encrypted data. In this setting, it is easy to see that $(\mathcal{E}, \mathcal{D}, \mathcal{H})$ is an enclosure scheme according to definition 1. Next, we provide the elements that make this scheme a safe enclosure according to definition 2.

7.1 Safety relation $\sim_{\mathcal{H}}$

Let us recall the IND-CPA (indistinguishability under chosen plaintext attacks) notion of security:

Definition 5 (IND-CPA [20, 21]). *An encryption scheme* $(\mathcal{K}, \mathcal{E}_0, \mathcal{D}_0)$ *is* IND-CPA *secure if for any* $(pk, sk) \in \mathcal{K}$*, any two messages* m_1, m_2*, and any polynomial time algorithm* $\mathcal{A}$*, the probability of the event* $\mathcal{A}[m_1, m_2, \mathcal{E}_0(m_i, pk)] = i$ *is negligibly close to* $\frac{1}{2}$*.*

When interpreted as a definition of data privacy, IND-CPA says that an adversary can not deduce any information about the plaintext given only the ciphertext and the public key. Can we reinterpret IND-CPA as a definition of server protection, by simply swapping the honest party and the adversary? Indeed, the definition says that no matter how an untrusted client chooses the plaintext, it does not make any difference to the server who only has access to encrypted data, no matter what polynomial computation is performed by the server. In particular, a plaintext that represents malware is indistinguishable from an innocent plaintext, when it is under encryption.

Therefore, we consider the following safety relation $\sim_{\mathcal{H}}$ for the enclosure scheme of this section: for any two messages m_1, m_2, we let $m_1 \sim_{\mathcal{H}} m_2$ if

and only if for any polynomial time algorithm $\mathcal{A}$, the probability of the event $\mathcal{A}[m_1, m_2, \mathcal{E}(m_i)] = i$ is negligibly close to $\frac{1}{2}$.

7.2 Enclosure protocol $(\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E})$

We rely on the generic enclosure protocol defined in section 5 and on the trusted enclosing transducer defined in section 6. For that, we have to show that the enclosure scheme satisfies homomorphic recomposability according to definition 3. In the role of atomic messages $a_1, \ldots, a_k$ we consider the bits 0 and 1 (alternatively, we could also consider the digits $\{0, 1, \ldots, 9\}$).

Now, the algorithm $\mathcal{A}_{\mathsf{dec}}$ consists in decomposing a message m into his binary representation $(m_1, \ldots, m_n)$. Conversely, given $(\mathcal{E}(m_1), \ldots, \mathcal{E}(m_n))$, the algorithm $\mathcal{A}_{\mathsf{rec}}$ consists in computing $\mathcal{E}(m_1) \star_\mathcal{E} \mathcal{E}(2^{n-1}) +_\mathcal{E} \ldots +_\mathcal{E} \mathcal{E}(m_n) \star_\mathcal{E} \mathcal{E}(2^0)$. Relying on the homomorphic properties of the underlying encryption scheme, we deduce $\mathcal{A}_{\mathsf{rec}}(\mathcal{E}(m_1), \ldots, \mathcal{E}(m_n)) = \mathcal{E}(m_1 \star 2^{n-1} + \ldots + m_n \star 2^0) = \mathcal{E}(m)$.

Then, as explained in section 5, the protocol for $\mathcal{U}_\mathcal{E}$ consists in applying $\mathcal{A}_{\mathsf{dec}}$ and sending the index $\langle a \rangle \in \{0, 1\}$ of atomic messages $a \in \{0, 1\}$ to $\mathcal{T}_\mathcal{E}$. It happens that $\langle a \rangle = a$ in our case. In turn, $\mathcal{T}_\mathcal{E}$ consists in instantiating the transducer $\mathcal{A}_\mathcal{E}$ from section 6 with inputs $\{0, 1\}$ and outputs $\mathcal{E}(0), \mathcal{E}(1)$, collecting the outputs from the transducer and recomposing $\mathcal{E}(m)$ relying on $\mathcal{A}_{\mathsf{rec}}$.

The following corollary is a direct consequence of propositions 1 and 2 and of the fact that the encryption scheme underlying $\mathcal{E}$ is IND-CPA:

Corollary 1. *$(\mathcal{E}, \mathcal{D}, \mathcal{H})$ is a safe enclosure scheme with respect to $(\mathcal{U}_\mathcal{E}, \mathcal{T}_\mathcal{E}, \sim_\mathcal{H})$.*

8 Open problems and future work

Security definitions. More research should be devoted towards security definitions that genuinely capture our adversarial model. We have instantiated the relation $\sim_\mathcal{H}$ from definition 2 with IND-CPA, for lack of better models at the moment. One way in which the security can be relaxed is to notice that a deterministic encryption scheme would be sufficient to implement a safe enclosure. Another thing that should be taken into account is the fact that the relation $\sim_\mathcal{H}$ should really depend on $\mathcal{H}$.

Enclosing schemes. Similarly, the enclosure function that we discuss in section 7 is only a first step. For instance, fully homomorphic encryption schemes could be revisited in this context and their efficiency could be improved in light of relaxed security notions. Homomorphic recomposability and enclosing transducers as defined in section 5 also deserve further research.

Abstraction refinement. The enclosing transducer that we propose in section 6 relies on a secure private channel abstraction. Obviously, the problem of a secure private channel is long standing in computer security, but our application context should open the way for new questions and solutions. Furthermore, there needs

to be more detail about how our framework would be actually implemented in practice. For instance, we assume that the outputs of the enclosing protocol can reach the server without being compromised. This protocol may be implemented on the server, on a proxy or on the client. Depending on where it lies, other questions can be asked. In any case, there needs to be a mechanism that separates the enclosing process from the server process, maybe based on a virtual machine monitor [22, 23].

Formal models and verification. Propositions 1 and 2 are proved at a rather high level of abstraction. We need to develop formal models that allow realistic and precise specifications of protocols and properties of interest. Ideally, we should be able to adapt standard methods and tools of protocol analysis to our new context, and rely on the significant progress in that area for automated verification [24].

References

1. Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.
2. Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, 2012.
3. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.
4. Trusted Computing Group. TCG Architecture Overview, Specification revision 1.4, 2007. www.trustedcomputinggroup.org.
5. Trusted Computing Group. TPM main specification, 2011. www.trustedcomputinggroup.org.
6. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
7. Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of np. *J. ACM*, 45(1):70–122, January 1998.
8. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Cynthia Dwork, editor, *STOC*, pages 113–122. ACM, 2008.
9. Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.
10. Victor Costan, Luis Sarmenta, Marten Van Dijk, and Srinivas Devadas. The trusted execution module: Commodity general-purpose trusted computing. *Smart Card Research and Advanced Applications*, pages 133–148, 2008.
11. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.
12. Joan G Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and Sean W Smith. Building the IBM 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.

13. Ross Anderson, Mike Bond, Jolyon Clulow, and Sergei Skorobogatov. Cryptographic processors - a survey. *Proceedings of the IEEE*, 94(2):357–369, 2006.
14. Ciaran Mclvor, Maire McLoone, and John V. McCanny. Fast montgomery modular multiplication and RSA cryptographic processor architectures. In *Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 379–384. IEEE, 2003.
15. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Chris Hankin and Dave Schmidt, editors, *POPL*, pages 104–115. ACM, 2001.
16. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.
17. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.
18. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.
19. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
20. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.
21. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 20(3):395, 2007.
22. Robert P Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.
23. Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.
24. Martín Abadi, Bruno Blanchet, and Hubert Comon-Lundh. Models and proofs of protocol security: A progress report. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.