DOI: https://doi.org/10.1016/j.future.2020.02.037

Blockchain-based Semi-Autonomous Ransomware

Oscar Delgado-Mohatar[1,*], José María Sierra-Cámara[b], Eloy Anguiano[1]

[a]*Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain*
[b]*Khoury College of Computer Sciences, Northeastern University, USA*

Abstract

Blockchain's benefits and advantages have been extensively studied in literature, but far fewer works can be found on the dishonest uses of them. In this paper, we present the first blockchain-based ransomware schemes, which use smart contracts and simple cryptographic primitives to provide a limited degree of automation and fair exchange. Specifically, the use of smart contracts would enable new capabilities for ransomware, such as the possibility of paying for individual files or the refund of the ransom to the victim if the decryption keys are not received within a specified period of time.

To demonstrate their feasibility, both technically and economically, these proposals have been implemented in the Ethereum Ropsten test network. The results show that running a full ransomware campaign similar to WannaCry, with more than 300,000 affected users, would have an additional cost of only 3 cents of a dollar per victim.

Finally, we show that there are no feasible countermeasures if these schemes are implemented in public blockchains. Therefore, we firmly believe that it is increasingly urgent to recognize and study this matter, in order to create new policies and technical countermeasures.

Keywords: blockchain, ransomware, smart-contracts

1. Introduction

The *blockchain* concept, first introduced as an auxiliary technology for Bitcoin in 2009 [1], has experienced a spectacular growth in the last years, with potential uses in almost every area of society [2]. However, every technology can be also used in an evil or criminal way and, blockchain is not an exception. This is compounded by the fact that new platforms, such as Ethereum [3], provide a much richer functionality than Bitcoin, through the support of smart contracts, based in very powerful scripting languages.

These functionalities may enable a plethora of new possibilities for cybercriminals. Juels et al. [4] introduces the concept of *criminal smart contracts*

*Corresponding author

(CSCs), and warns about their potential to leak confidential information, key theft, or even the facilitation of real-world crimes, such as murder or terrorism.

In this paper, we present and analyze a new possibility: *the implementation of a semi-autonomous ransomware infrastructure coded as a smart contract*. The benefits of this approach for the criminals would be numerous, specially those regarding *reliability*, with a platform virtually immune to authorities and shutdown.

In addition, the use of smart contracts would provide to ransomware new ways to interact with victims. For example, victims could pay by the decryption of individual files, with dynamic prices marked by attackers, depending on the type of file or other factors. The victims could also have more guarantees about the ability of the attacker to decrypt their files trough a proof-of-life mechanism, in which the victim could chose a small subset of files to be decrypted for free. We believe that all these possibilities would increase the willingness of the victim to pay the ransom and, in turn, finally benefit the attackers.

In order to study the real viability of these ideas, we have implemented a proof-of-concept in the Ethereum platform, including the different possibilities which we foresee could be used by attackers. To keep them as simple as possible, and minimize their associated execution and storage costs, these schemes use only symmetric cryptographic primitives, basic arithmetic operations and data storage in arrays.

Therefore, the key questions we explore in this work are: could a smart contract-based ransomware be fully implemented in a public blockchain? How practical this would be? Which would be the associated costs? And, as a result of the answers to these questions, should this idea be considered a potential threat to blockchain's future?

Contributions

Our specific contributions in this work are:

- We present the first architecture of ransomware based in the use of a public blockchains and smart contracts. In order to minimize its execution costs, the design is kept as simple as possible, and based only in symmetric cryptographic primitives.
- We demonstrate its feasibility by implementing a simple but functional proof-of-concept in Ethereum, and analyzing its execution costs.
- We introduce new ransomware payment paradigms, enabled by the use of smart contracts, as *pay-per-decrypt* or *proof-of-life*, which provide limited fair-exchange capabilities.
- We discuss some possible mitigation countermeasures.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to blockchain and ransomware technologies, including some figures about its current prevalence in cyberthreats. Section 3 introduces the concept of blockchain-based ransomware, and its general working principles and characteristics. Section 4 presents three novel protocols, *pay-and-pray*, *pay-per-decrypt* and *proof-of-life*, along with their main characteristics and an analysis of their associated storage and execution costs in Section 5. A discussion about mitigation and countermeaseures can be found in Section 6. Finally, the conclusions are presented in Section 7.

2. Background

2.1. Blockchain basics

Despite its enormous potential, the *blockchain* concept has a modest and recent origin. As defined today, it was firstly described as an auxiliary technology of Bitcoin in 2009 [1][5], where it is used as a secure mechanism to store economic transactions between participants. Its recent explosion in popularity is due to the possibility of also securely storing any kind of digital data, guaranteeing its integrity.

This automatically enables many new possible uses for the technology: *certification of documentation* (as mortgages, securities or any other official document [6][7]), *assets or intelligent objects* [8][9], which can make decisions based on the information stored in the blockchain, a *distributed securities market*, *deposit and custody services* [10], which would resolve disputes between customers and merchants, *voting systems* [11][12] or improvements in the *supply chain* for all types of products [13][14][15].

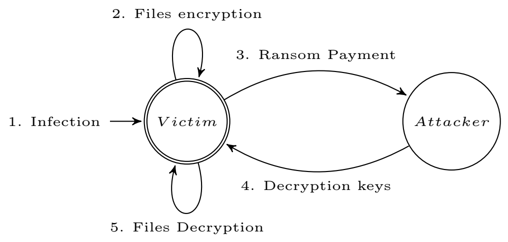
2.2. Ransomware basics

The use of cryptography to mount extortion-based attacks was first proposed by Young and Yung as early as 1996 [16]. In their seminal paper, they actually created a new research area called *cryptovirology.* In their initial proposal, which remains almost unchanged, the attacker places a public/private pair key in a malware. The malware then encrypts the victim's files with a locally generated random symmetric key, which is in turn encrypted with the public key. Finally, it provides a ransom note and instructions for payment. After the attacker receives the payment, the previous symmetric key is decrypted with the corresponding private key and sent to the victim, who finally recovers the original files.

Amazingly, the first reactions to this potential threat were skepticism and criticism. They were even denominated as "virtually useless". The fact is that, in the recent years, ransomware has become one the biggest threats to information security, with almost daily news about its impact [17][18].

WannaCry phenomenon. Maybe the most known recent example is the WannaCry ransomware campaign, which became a global phenomenon in May 2017, and has probably been the most significant security incident in recent years. It had world-wide diffusion, affecting more than 300,000 computers in about 150 countries, and cost thousands of dollars in ransom to regular users and millions to enterprises in lost productivity [19].

Figure 1: Common ransomware scheme, including infection, payment and release of keys



Today, scientific research about ransomware goes far beyond the mere study of the infection and encryption process. For example, some studies analyze the reasons and circumstances that most influence the willingness of victims to pay [20], the estimated amount of money that ransomware has raised so far [21], or attacks in new environments, such as mobile phones [22][23]. In this sense, our study opens a new perspective, by proposing the implementation of ransomware by using automated smart contracts.

2.3. Ransomware spreading and infection

Ransomware, like any other form of malware, is delivered to the victims by using common *attack or infection vectors*. These include Web, P2P networks and email, in order of prevalence [24]. The exact infection procedure varies slightly for each infection vector, but a common scheme can be found in Figure 1.

In brief, the usual steps in the transfer and infection procedure are the following:

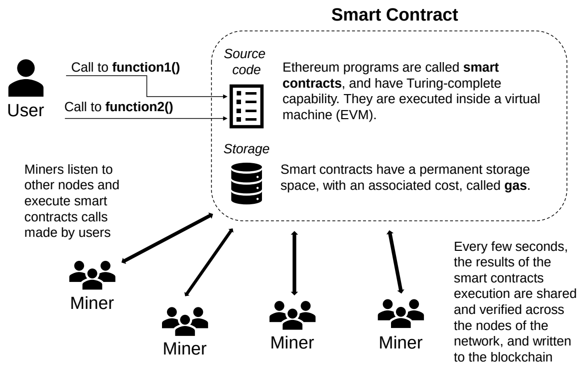
1. **Setup**: The cybercriminals implement and get ready all the elements needed for the attack. For example, for an infection via Web, they would need to previously compromise a Web server (the more visited, the better), in which they install an exploit code.

2. **Infection**: From this point, each user visiting the Web page with a vulnerable browser gets infected. The process usually includes a small code snippet, called *dropper*, that downloads the rest of the ransomware binary.

3. **Files encryption**: the ransomware encrypts all the files of interest in the victim's computer.

Bitcoin as a payment method. Cryptocurrencies provide unique privacy advantages for criminals behind ransomware attacks over traditional payments methods. For example, Bitcoin transactions are cryptographically signed messages which embody a fund transfer from one public key to another. Furthermore, Bitcoin keys are not explicitly tied to real users, although all transactions are public. Consequently, ransomware owners can protect their anonymity and avoid revealing any information that might be used for tracing them [25] [26] [20].

Figure 2: High-level architecture of Ethereum, where main concepts of smart-contract, storage, miners and functions are depicted.


Smart Contract
User
Call to **function1()**
Call to **function2()**
Source code
Ethereum programs are called **smart contracts**, and have Turing-complete capability. They are executed inside a virtual machine (EVM).
Storage
Smart contracts have a permanent storage space, with an associated cost, called **gas**.
Miners listen to other nodes and execute smart contracts calls made by users
Miner
Miner
Miner
Miner
Every few seconds, the results of the smart contracts execution are shared and verified across the nodes of the network, and written to the blockchain

2.4. Blockchains and smart contracts

Blockchain was initially proposed as an auxiliary technology for Bitcoin, which has solved for the first time, in an effective and decentralized way, the old anonymous e-cash challenge [27].

On the other hand, Bitcoin introduced the use of *proof-of-works* (PoW) [28] as part of a distributed consensus protocol. This way, a Blockchain is able to verify transactions and maintain a global ledger, shared by each node (i.e., *miner*).

Finally, Bitcoin also offers a limited support for a programmable logic, trough the use a scripting language. However, this feature has proven to be restrictive and difficult to use, as demonstrated by previous attempts to build more complex applications using Bitcoin [29][30][31] .

To overcome these limitations, blockchains with much more powerful computation capabilities have been developed, such as Ethereum [32][33]. This platform, for example, provides a Turing-complete logic, which enables the implementation of general-purpose smart contracts.

Figure 2 shows a high-level architecture of a platform like Ethereum. In this case, smart contracts are included in transactions, and are verified in a similar way to Bitcoin. This way, nodes can reach a consensus, not only about data, but also about the results of a computation, through the correct execution of contracts logic.

In addition, due to its intrinsic immutability, smart contract execution cannot be blocked by anyone, even its creator. This will have important consequences for blockchain-based ransomware, as we will analyze in section 6.

Ether and gas. To encourage the execution of these contracts, Ethereum has its own associated cryptocurrency, called *Ether*. Each operation, including both computation and data storing, has a fixed cost, measured in a unit called *gas*. Gas is paid in Ether, which, in turn, can be exchanged by fiat currency.

The execution of a contract can fail for many reasons, but it is important to note that, even in that case, the executor must still pay the proportional fee to miners.

2.5. Blockchain and crime

Almost since its inception, Bitcoin has always been associated to money laundering and illicit activities. Indeed, there exist studies of crime enabled by Bitcoin, such as the infamous Silk Road marketplace [34] or money laundering [35]. It is remarkable, in any case, that these studies also show that money laundering is not as massive as Bitcoin critics claim, due to automation mixes, necessary for this activity, cannot operate on large enough volumes to be significant on a world scale.

As a first example of other potential misuses of blockchains, we show the case of the Darkleaks protocol for leakage of secrets [4]. Darkleaks is a decentralized black market for selling and buying (usually leaked) information.

The main objective of Darkleaks is to provide a solution to the *fair exchange problem* between trust-less parties, which is described with more detail in section 3.1. Particularly, in this case, we are dealing with *fair payment* (FP), in which one of the parties wants to be sure to pay for a commodity only if it is received, and the other to send the commodity only if the payment can be assured in advanced [36].

With documents, or digital goods in general, there is another shortcoming: how to prove the authenticity of the document *before* the transaction? To do so, Darkleaks uses a trust-less probably fair mechanism. First, the leaker splits the file to be sold into a number of chunks, which are hashed. Then, each hash becomes a Bitcoin private key, from which a public key and an address are derived. Finally, each chunk is encrypted with a secret obtained by hashing the public key.

Then, when the public sale of the file begins, the leaker randomly selects some of the previous chunks, which will be released free of charge. This allows the community to verify the veracity of the file and decide whether they want to pay for the remaining segments.

If so, the buyers send payments to each segment address, till some threshold quantity is reached. Then, when the leaker decides to claim it, the public key is revealed, which, in turn, means that the encryption key for that chunk is also implicitly revealed (because one is derived from the other).

2.6. Threat Model and Security Guarantees

Roughly speaking, we adopt a common thread model, as presented in [4]:

- *Blockchain.* Trusted for correctness but not privacy. We assume that the blockchain will always correctly store data and perform computation, and will always remain available. However, the blockchain exposes all of its internal states to the public, and retains no private data.

Concept	**Description**	**Concept**	**Description**
$\mathcal{A}$	attacker, ransomware creator	msk_v	master key for victim $\mathcal{V}$
$\mathcal{V}$	victim	id_v	unique identifier for victim $\mathcal{V}$
$\mathcal{C}$	smart contract	id_f	unique identifier for file f
dk_v	decryption symmetric key for all the encrypted files of victim $\mathcal{V}$	$h()$	hash function
dk_f	decryption symmetric key for a specific encrypted file f	$enc_k()$, $dec_k()$	encryption and decryption functions with symmetric key k

Table 1: Notation for main concepts and parties

- *Arbitrarily malicious parties.* We assume that the parties involved do not trust each other, and that they act solely to maximize their own profits. In particular, they may arbitrarily deviate from the prescribed protocol and prematurely abort the protocol.

- *Network influence of the adversary.* We assume that messages exchanged between the blockchain and parties are guaranteed to be delivered within bounded delay. However, an adversary can arbitrarily reorder these messages.

2.7. Notational Conventions

The notation for parties and concepts involved in this work are shown in Table 1.

3. Exchange Fairness and Ransomware Automation

As stated before, one of the biggest concerns for ransomware victims is the lack of guarantees that, even if ransom is paid, they actually receive the decryption keys. Indeed, this situation has already happened on numerous occasions in the past, and may be due to multiple factors, such as scalability problems (impossibility of manually attending to hundreds of thousands of victims, as in the WannaCry incident), infrastructure shutdown by authorities or, simply, a fake ransomware campaign, similar to fake DDoS threats [37].

In this section, we analyze the possibilities for attackers to design more trustworthy ransomware schemes and, therefore, improve the willingness of the victim to pay the ransom. For example, attackers could try to provide a limited fair exchange capability by making make payments conditional on the delivery of the correct encryption keys. Taking advantage that almost all of ransomware employs some cryptocurrency as a payment mechanism (typically, Bitcoin), a ransomware developer could implement a script (such as an Ethereum smart contract, for example) to guarantee the following property:

The requested payment will be delivered if and only if the attacker reveals a correct decryption key.

However, the previous idea does not address the problem of scalability for attackers. So, could the ransomware schemes be automated in such a way that they don't require operators or manual intervention?

This property is not as feasible to accomplish as fairness, because smart contracts are usually run in blockhains where all their executions and states are public. Therefore, they could not store or manage the necessary (secret) decryption keys. In theory, zero-knowledge techniques could be applied in this situation, but not in the ransomware scenario. The reason is that, as will be described in upcoming sections, it is not possible for a smart contract (or any other program) to automatically determine if a file really belongs to a user, even if a correct decryption key has been provided.

In summary, from the point of view of an attacker, an ideal ransomware would have the following two properties:

- **Exchange fairness**: the payment should be delivered to the attacker if and only if the user receives a correct decryption key.
- **Maximum automation degree**: ideally, the ransomware should not need human operators to process payments and release decryption keys, and to operate autonomously.

In the following sections, we study these possibilities, and analyze to what extend they can be achieved. Finally, in section 4 we present several protocols which implement these ideas.

3.1. Exchange fairness

The fairness in the exchange of items or services is a corcern that naturally arises in a digital scenario such as the Internet and e-commerce. The fundamental question is how to sell an item in such a way that none of the involved parties can cheat the other [38].

The classical definition of exchange fairness includes two parties, Susan and Bob, who act as Seller and Buyer, respectively. In this setup, Susan wants to sell some kind of digital good x to Bob. Obviously, Bob does not know x, but he can impose some restrictions over it.

For example, he could be define a predicate $f : 0,1^* \rightarrow \{true, false\}$, and be willing to pay Susan for all the values of x which satisfy $f(x) = true$. f is usually written as a computer program, and will vary depending on the nature of x. Typically, if x is some kind of file (e.g., a movie), $f(x)$ would output *true* only if the hash of x corresponds to some fixed value h.

Of course, the problem that arises now is: if the seller and buyer do not trust each other, who should initiate the transaction and how should it be done?

The protocols and mechanisms designed to manage this situation are called *fair exchange* schemes. Although there is a wide variety of protocols with different capabilities, there are some desirable common characteristics that all fair exchange protocols should provide:

- **Fairness**: intuitively, a protocol is fair if at the end of its execution there are only two possible states: either all participants receive the expected item correctly, or none of them received what they expected. In other words, the protocol does not allow one user to take advantage of the other.

- **Timeless**: guarantees that any user can abort the protocol at any time, without the need for the participation of the other user.
- **Completeness**: the protocol is considered complete if all the honest parties finally get all the desired data from the others participants.

However, it is well-known that all these properties for a fair exchange are impossible to achieve without a trusted third party (TTP) in real environments [39]. In this scenario, blockchains can play the role of this trusted entity or arbiter, with the added advantage that blockchain (specifically, its associated cryptocurrency) is already being used to make the payments between the parties.

In a first approach, this can be achieved by using a smart contract C on a public blockchain with semantics similar to the following:

1. The buyer sends to C some amount of cryptocurrency, e.g., 1 BTC, staying in custody.
2. The seller claims this amount by sending x such that $f(x) = true$ to C. The smart contract then verifies that x is actually a valid value and, if so, sends the agreed amount to the seller.
3. If, for any reason, the seller does not claim payment in a period of time t, C automatically sends the money back to the buyer.

Indeed, the basic primitive suitable for achieving this in Bitcoin already exists and is called *Zero Knowledge Contingent Payment (ZKCP)*, which was proposed by Maxwell [40] as early as 2011. However, this initial proposal had to wait almost 5 years more to become powerful enough to perform general purpose ZKCPs. Finally, ZKCPs were able to be demonstrated live by Bowe from ZCash Team [41][42], swapping the solution to a 16x16 sudoku puzzle for 0.1 BTC.

3.2. Zero Knowledge Contingent Payment (ZKCP)

The main idea behind the ZKCP protocol is actually very simple: the seller (Susan) generates a *zero-knowledge proof* of the item x and a transaction to redeem the funds only if the buyer (Bob) can verify this proof successfully.

Specifically, Susan uses a symmetric encryption algorithm *enc* with a key k to encrypt x, such that $enc_k(x) = c$. She also uses a hash function h to compute $h(k) = y$. She then sends these values c and y to Bob, together with a zero-knowledge proof that c is the ciphertext of x under the key k and that $h(k) = y$.

Once the proof has been received and verified by Bob, he creates an smart contract in a pubic blockchain which pays to Susan the agreed amount if she provides a valid value for k, such that $h(k) = y$. Bob, who can recover k from the smart contract, can finally decrypt c and recover x.

3.2.1. Application to malware

At first glance, these ideas seem directly applicable to the automation of ransomware. This must only choose an encryption key k for each file, and leave a public hash value $y = h(k)$ in the victim's system. It is, then, not difficult to write a script in Bitcoin, by using the ZKCP construction, or a smart contract in Ethereum, that allows payment if and only if the attacker is able to post the pre-image of y.

Similarities with the usual uses end here, however. The challenge for the ransomware scenario is to prove that k is actually a valid decryption key, and that the victim is able to recover her files with it. One possible solution to this problem is to use *zero-knowledge proof schemes*, like zkSNARKS [43][44], which would make the process non-interactively. Indeed, Ethereum has recently successfully completed the first integration of this technology, and has added the arithmetic modular primitives necessary for its implementation.

However, even this primitive would not be enough to solve the problem in the case of ransomware. The reason is that is not possible to write an automatic code, inside or outside a blockchain, that verifies that a file is legitimate for a user. Even if it automatically detects, for example, that it is a *semantically* valid PDF file (e.g., applying strict format checks), it is impossible to know if it is actually the user's PDF.

After all, taking into account that the type of files targeted by malware is known and limited (word-processing files, videos, photos, etc.), the malware could use the strategy of replacing the files with others "syntactically" correct but semantically empty. That is, it could replace the MP3 files with others containing simply noise, the PDFs with others with blank pages, etc ...

These files would still decrypt successfully, but they would not correspond to the original data of the user, which is the only outcome that she cares about. Note that in this case authenticated encryption or other kind of integrity checks neither help.

3.3. Ransomware automation

As it is clear, it is not possible to fully automate the ransomware process, since the "semantic" verification of the decryption of the files must always be carried out, ultimately, manually by the user. A possible solution would be to provide the user with a kind of oracle that could examine a file (with all its metadata, such as complete path, length, timestamps, etc.) and determine if it really belonged to the user initially. In practice, this could be essentially done in two ways:

- Collection of digital signatures on the files, which could be verified later.
- Manual inspection of a small subset of files.

Unfortunately, the first possibility doesn't seem realistic, due to that the collection should be created in advance, *before* the ransomware attack. In addition, it should be keep updated in each modification of the files. For these reasons, a *statistical approach* would be more appropriate for the second option, by interactively verifying the encryption process.

The idea is actually very simple: the attacker allows the user to decrypt some files free of charge (or for a small fee), in a kind of *proof of life*. If the files decrypt successfully to the semantically correct files, it is reasonable to assume that the attacker knows the appropriate decryption keys.

For the attacker interest, this subset should: (i) be selected randomly (to prevent the victim from deciphering the files of most interest to her and leaving the rest), and (ii) account for a small percentage of the total number of encrypted files.

4. Semi-autonomous Blockchain-based Ransomware

In this section we present several novel protocols for implementing blockchain-based ransomware schemes. We foresee their use in the wild soon, due to the advantages they provide to cybercriminals over the traditional schemes.

Essentially, these schemes would use smart contracts as a payment escrow service: the smart contract acts as a judge who withholds payment from the victim's ransom until the attacker reveals a correct decryption key. If a certain amount of time elapses and the attacker has not disclosed a valid decryption key, the smart contract automatically returns the payment to the victim.

As stated before, this approach would have important benefits from an attacker's perspective, as an increase of the resilience of the overall system, and new properties as *fair exchange* or *partial automation*. In addition, it would also provide victims with greater guarantees than in traditional ransomware schemes and, therefore, increase their likelihood of paying the ransom.

Finally, the use of smart contracts would provide even more additional benefits for attackers:

- The scheme can be entirely designed to use symmetric-key primitives only. This allows to keep it as simple as possible, and to minimize the execution costs for the smart contracts.
- The resilience of the ransomware infrastructure is increased. Once a smart contract is deployed to a blockchain, it is extremely difficult to remove it, even after being identified as malicious. We will present some possible solutions in section 6.
- The possibility of establishing a kind of *ransomware-as-a-service*. This way, attackers could rent the use of the smart contract to others cybercriminals, who would make use of the same infrastructure to launch different ransomware campaigns.

By using these premises, we envision the following possible schemes, ordered in increasing complexity and capabilities:

- **Pay-and-pray**: the victim has no special guarantees, and pay for the decryption of all her files as a whole.

- **Pay-per-decrypt**: the victim can pay for the decryption of individual (chosen) files.
- **Proof-of-life**: The victim has the guarantee of a limited fair exchange, that is, to be able to recover her money if the attacker finally does not deliver a valid key for the decryption of the files.

Although these differences, the protocols share a common initial step:

1. **Setup**. Prior to the launch of a ransomware campaign, the attacker generates a random identifier, id_v and a master secret key, msk_v, both unique for each victim. These values are embedded in a customized executable E_v.

   As a result, the attackers creates and protects a database $\mathcal{B}$ containing all the previous tuples (id_v, msk_v). They will later used to reveal the appropriate decryption key from the identifier provided by a victim.

The aim of this setup phase is to avoid the necessity of a posterior communication with the attacker to send a locally generated decryption key. In traditional ransomware schemes, which use asymmetric primitives, this step is performed by encrypting the symmetric decryption key with the attacker's public key.

These schemes are discussed in more detail below.

4.1. Pay-and-pray

As a first approach, this scheme would be conceptually similar to the existing ransomware models, where there is not any kind of guarantee for the victims after payment. However, the attackers would still benefit from transferring part of their infrastructure to a blockchain, thus increasing their resistance.

Informally, the protocol can be summarized in the following steps:

1. **Setup**. Performed as described in the previous section.
2. **Infection and files encryption**. After infection, the ransomware executable E_v encrypts victim files with a single symmetric key, derived from her identifier and the victim master key as $dk_v = h(id_v||msk_v)$. It also calculates a commitment of the decryption key, $c = h(dk_v)$, that will be used by the victim later. After this step, msk_v is securely wiped, and the values id_v and c remain stored in the victim's system.
3. **Payment**. The victim pays the requested amount to the smart contract $\mathcal{C}$, including in the request the values of the identifier id_v and the commitment c created by the attacker in the previous step.
4. **Disclosure**. Attacker retrieves id_v from the contract, and reveals the corresponding msk_v. The contract, in turn, checks this value against the previously stored commitment and, if it is correct, pay the victim's ransom to the attacker.

5. **Key collection**. Victims can periodically verify if their decryption keys have been revealed. If a certain amount of time has elapsed without successful releasing, the ransom is refunded to the victim.

Of course, nothing forces an attacker to respect the terms of step 1, where supposedly a commitment of the decryption key is calculated. The attacker could just fake this value, along with the master key, fooling the smart contract in the **Disclosure** step. However, attackers have no incentive to do this, since the victims would quickly stop paying ransoms when the first cases were publicly known.

In any case, this protocol would still provide both victim and attackers some advantages over traditional schemes:

- The victims are guaranteed by the smart contract that they will recover the ransom if no keys are revealed within a period of time.
- The attackers have a more resilient infrastructure, almost impossible to be shutdown by authorities.

The contract. The smart contract pseudo-code for this protocol is shown in Figure 3. It has been implemented using Ethereum [32], Solidity language [3] and deployed into the Ropsten Ethereum testnet. More details and the source code can be found in Appendix 8.

A flowchart for the protocol can be found in Figure 4. Attacker and victim interact by alternately calling the appropriate smart contract functions:

0. **init()**. Attacker deploys smart contract $\mathcal{C}$ to a public blockchain (e.g., Ethereum), and initializes it by calling *init()* function with a ransom amount. This amount could be even be changed dynamically during the attack campaign.

1. **Infection and files encryption**. After infection and files encryption, the victim identifier and the decryption key commitment, id_v and c, are generated and stored in the victim's system.

2. **payRansom**(amount, id_v, c). The victim decides to pay the requested amount by calling this function, with the values (id_v, c) as arguments.

3. **getVictims()**. Attacker periodically calls this function, which returns a list of victims who has already paid the ransom. For each victim with identifier id_v, the corresponding master secret key msk_v is recovered from database $\mathcal{B}$. Finally, decryption key for each victim is derived as $dk_v = h(id_v||msk_v)$.

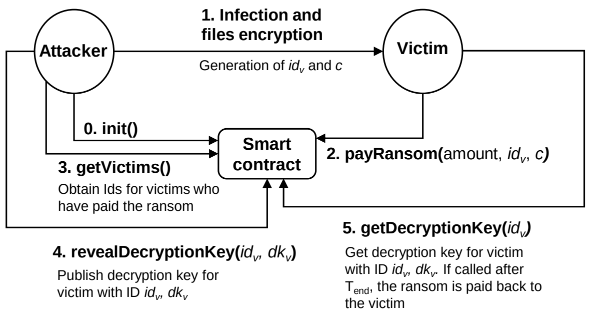
4. **revealDecryptionKey**(id_v, dk_v). Then attacker calls this function for each victim, with the decryption key calculated in the previous step as an argument. The contract $\mathcal{C}$ then verifies that the commitment of each one of them is correct and release the payment made by the victim.

Figure 3: Pay-and-pray smart contract pseudo-code

```
Initially deployed to Ethereum blockchain by an attacker $\mathcal{A}$.
Functions called by attacker
init(owner_address, min_ransom)
    · Set smart contract owner address to attacker's, owner_address := $\mathcal{A}$
    · Set minimum ransom amount to be paid to min_ransom.
    · Set array victims_ID := {}, containing the IDs of the victims who have
      already paid the ransom.
getVictims()
    · Return victims_ID.
revealDecryptionKey($id_v, dk_v$)
    · Assert victim.commitment $= h(dk_v)$.
    · Set victim.state := REVEALED
    · Set victim.decryptionKey := $dk_v$
    · Set ledger[$\mathcal{A}$] := ledger[$\mathcal{A}$] + $amount

Functions called by victims
payRansom(amount, $id_v$, $c$)
    · Assert that $amount ≥ $min_ransom.
    · Set victims_IDs:= victims_IDs + $id_v$.
    · Set victim.id:= $id_v$, victim.commitment:= $c$.
getDecryptionKey($id_v$)
    · Assert victim.state = REVEALED
    · If victim.state = PAID and T > $T_{end}$:
        · Set ledger[$\mathcal{V}$] := ledger[$\mathcal{V}$] + $amount
    · Return victim.decryptionKey
```

Figure 4: Pay-and-pray scheme architecture


Attacker
Victim
1. Infection and files encryption
Generation of id_v and c
0. init()
Smart contract
2. payRansom(amount, id_v, c)
3. getVictims()
Obtain Ids for victims who have paid the ransom
4. revealDecryptionKey(id_v, dk_v)
Publish decryption key for victim with ID id_v, dk_v
5. getDecryptionKey(id_v)
Get decryption key for victim with ID id_v, dk_v. If called after T_{end}, the ransom is paid back to the victim

5. **getDecryptionKey(id_v)**. Finally, each victim periodically calls this function, which returns the revealed key to its owner. If the decryption key has not been revealed before the specified deadline (T_{end}), the ransom is refunded to the victim.

Notice that there is no security concerns with the public revelation of id_v and dk_v, because none of the values are useful for other parties:

- An user does not benefit from using another user's identifier, since he would obtain incorrect decryption keys for himself.
- The decryption key published by the attacker is not valid for another victim, nor to decrypt other files of the legitimate user.

On the other hand, the use of smart contracts would ease to dynamically adjust both min_ransom and T_{end}. In the first case, for example, the amount of the ransom may increase over time, to encourage victims to pay quickly. A similar concept could be applied to T_{end}, so that attackers could establish different ransom amounts to be paid based on T_{end}. The lower the T_{end}, the higher the min_ransom and vice versa.

4.2. Pay-per-decrypt

The use of a blockchain and smart contracts enable a new possibility for attackers. We have called it *pay-per-decrypt*, and it provides the user with the capability to pay for the decryption of individual files, instead of the full set. The rationale of this is that, this way, many victims who are reluctant to pay the entire ransom, are encouraged to pay smaller amounts for a file, or only for those they really need to recover.

In addition, this approach allows victims to gain trust on the scheme: if the recovery of the first file is successful, it is reasonable to assume that the rest will be too, because the attacker would cease to make a profit otherwise. And if it is not, the loss for the victim would be much less than in the *pay-and-pray* approach.

Finally, this scheme allows the attacker to establish several payment conditions, such as different prices depending on the type of file, discounts for volume, etc.

In brief, the protocol is composed of the following steps:

1. **Setup phase**. Same as described previously.
2. **Infection**. In this case a different decryption key is used for each user file. In this way, for each file f a random ID, id_f, is generated. From this value, a symmetric decryption key $dk_f = h(msk_v, id_v||id_f)$ and a commitment $c_f = h(dk_f)$ are calculated. All these values are locally stored in the victim system, along with the encrypted files.
3. **Payment**. The victim chooses a file to be decrypted, and pays the associated ransom by calling the smart contract function *payRansom()* with

arguments id_v, id_f and c_f, the ID of the victim, the ID of the file to decrypt and the commitment for that file, respectively.

4. **Disclosure**. Attacker now performs the following operations:

   - The list of tuples (id_v, id_f) for the victims who have paid the ransom is retrieved by calling *getFileIDs()*.
   - For each tuple, the corresponding decryption key $dk_f = h(msk_v, id_v || id_f)$ is derived from the previous values.
   - A new list of tuples (id_f, dk_f) is revealed to the smart contract by calling *revealDecryptionKeys()*. The smart contract verifies then each stored commitment against each revealed key, checking that $c_f = h(dk_f)$.

5. **Key retrieval**. Victims can periodically verify if their decryption keys have been revealed by calling *getDecryptionKey(id_f)*. Again, the victim payment is refunded automatically by the smart contract if attacker does not provide decryption keys within a certain period of time.

The smart contract pseudo-code and a general view for the architecture can be found in Figure 5 and Figure 6, respectively. Of course, in a real environment, the previous operations could be optimized in several ways. For example, to avoid the polling of the function *getDecryptionKey()* by the victim, the smart contract could use *events* and an external oracle to notify the victim via email that their decryption keys have been revealed.

4.3. Proof-of-life

Finally, in this section we present the last scheme that, in our opinion, could be used by attackers. This scheme takes into account all the previous considerations, and tries to overcome some of its limitations.

The main improvement is the addition of an interactive "proof of life" procedure, based in a challenge/response scheme. In this way, the user or the smart contract can choose a small subset of files to be decrypted "for free". The objective is to increase the victim's trust in the scheme, and her willingness to pay for the rest of her files.

The protocol is composed by the following steps:

1. **Setup phase, Infection**. Same as described for the *pay-per-decrypt* scheme, i.e., a different decryption key is generated for each file. However, in this case only one commitment is generated for the victim's master key, $c = h(msk_v)$.

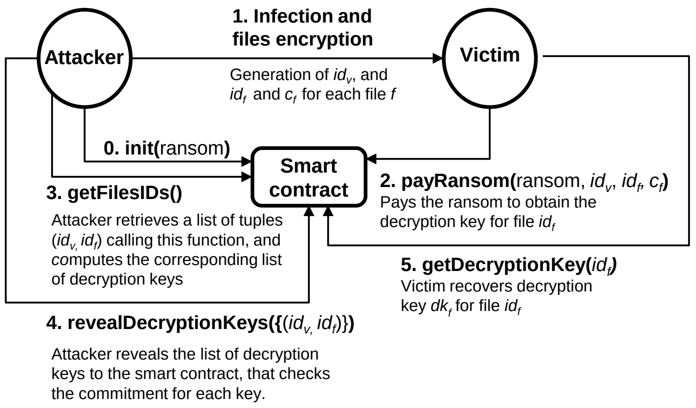
2. **Challenge**. In this step, a *proof of life* procedure is performed before payment. To do so, the victim is allowed to choose a small subset of k files that will be decrypted *for free*. Of course, k will be determined by the attacker in a small value (maybe a percentage of the victim's total number of encrypted files). Now, the following actions are executed:

Figure 5: Pay-per-decrypt smart contract pseudo-code

```
Initially deployed to Ethereum blockchain by an attacker $\mathcal{A}$.
Functions called by attacker
init(owner_address, min_ransom)
    · Set smart contract owner address to attacker's, owner_address := $\mathcal{A}$
    · Set minimum ransom amount to be paid to min_ransom.
    · Set array files_ID := {}, containing the IDs of the files whose ransom
      has been paid.
getFilesID()
    · Return files_ID.
revealDecryptionKeys($\{(id_v, dk_f)\}$)
For each tuple $(id_v, dk_f)$:
    · Assert victim.fileID.commitment = $h(dk_f)$.
    · Set victim.fileID.state := REVEALED
    · Set victim.fileID.decryptionKey := $dk_f$
    · Set ledger[$\mathcal{A}$] := ledger[$\mathcal{A}$] + $amount

Functions called by victims
payRansom($amount, id_v, id_f, c_f$)
    · Assert that $amount ≥ $min_ransom.
    · Add victim's ID to paid_IDs array.
    · Set victim.fileID.commitment := $c_f$
getDecryptionKey($id_f$)
    · Assert victim.fileID.state = REVEALED
    · Return victim.decryptionKey
    · If victim.fileID.state = PAID and T > $T_{end}$:
        · Set ledger[$\mathcal{V}$] := ledger[$\mathcal{V}$] + $amount
```

Figure 6: Pay-per-decrypt scheme arquitecture


Attacker
Victim
1. Infection and files encryption
Generation of id_v, and id_f and c_f for each file f
0. init(ransom)
Smart contract
2. payRansom(ransom, id_v, id_f, c_f)
Pays the ransom to obtain the decryption key for file id_f
3. getFilesIDs()
Attacker retrieves a list of tuples (id_v, id_f) calling this function, and computes the corresponding list of decryption keys
5. getDecryptionKey(id_f)
Victim recovers decryption key dk_f for file id_f
4. revealDecryptionKeys({(id_v, id_f)})
Attacker reveals the list of decryption keys to the smart contract, that checks the commitment for each key.

(a) The victim calls the *requestSamplesDecryption()* function with a list $\{id_f^*\}$ containing the IDs of the chosen files as an argument. The superscript * indicates that this is a free-decrypted file.

(b) The smart contract checks that the length of this list is less than k and mark these files as "decrypted".

3. **Response**. The attacker now responds to the previous challenge in a similar way to the *pay-per-decrypt* scheme:
   - The list of tuples $(id_v, \{id_f^*\})$ is retrieved by calling *getFileIDs()*.
   - For each tuple, the corresponding decryption keys $dk_f^* = h(msk_v, id_v||id_f^*)$ are calculated. There will be a maximum of k free decryption keys.
   - A new list of tuples $(id_f^*, \{dk_f^*\})$ is revealed to the smart contract by calling *revealSamplesDecryptionKeys()*.
4. **Payment**. The victim retrieves these keys by calling *getFreeDecryptionKeys(id_v)*. If satisfied with the result, she pays the requested amount to the smart contract with the call *payRansom(id_v, c)*.
5. **Disclosure**. In the final step, the attacker reveals msk_v to the contract, that verifies its validity by checking that the set $\{dk_f^*\}$ can be successfully derived from it. With this master key, the victim can now calculate the decryption key for any remaining file. As in previous schemes, if after a specified period of time, the attacker doesn't reveal msk_v, the contract gives the money back to the victim.

In addition, the attacker might also use an alternative approach by letting the smart contract to randomly choose the files to be freely decrypted, instead of the victim.

In this case, the protocol should be slightly modified in order to assign the IDs of the encrypted files sequentially in the infection phase. After that, the function *requestSamplesDecryption()* is no longer called by the victim, and could be substituted by a code similar to shown in Figure 7. There exist several ways to generate random numbers in a smart contract, but we propose here a simple one based on the previous block hash and the SHA3 function. This smart contract pseudo-code can be found in Figure 8, and a global view of the scheme is shown in Figure 9.

4.4. Analysis of the C/R stage

As stated in the previous section, the main purpose of the challenge/response stage is to increase the victim's confidence that the attacker has valid decryption keys for all the files (or, at least, most of them).

The natural question is, then, can be this confidence degree measured? The answer depends on who, victim or attacker, chooses the files to be sampled.

Figure 7: Generation of random numbers for the election of free-decryption file IDs.

```
uint seed = block.blockhash;
function rand(uint min, uint max)
                    returns (uint){
    seed = sha3(seed);
    return uint(seed)%(min+max)-min;
}
```

If the victim is allowed to choose freely, she obviously would always choose the files she cares most about first, possibly losing the interest in paying for the rest. The attacker could avoid this by also encrypting the file and directory names, so the user cannot distinguish which are the files of most interest.

But, even if the victim has no other option than to select the challenges randomly, the theory of probabilities favors her. Indeed, even a small number of successful decryptions provides high confidence against cheating.

For estimating this, let n_t, n_f, and n_s be the total number of files, number of lost or modified files by the attacker, and the number of allowed samples, respectively. Then, the probability of the user to detect an attacker cheating attempt, p_c, is:

$$p_c = 1 - \prod_{i=0}^{n_s-1} \frac{(n_t - n_f) - i}{n_t - i}$$

For example, for a total number of 10,000 files, supposing than only 2% were manipulated and that the user is allowed to check 60 files for free, the final probability of detecting the fraud is more than 70%.

5. Experimental Results

In this section, the results of the experiments and implementations carried out are presented and analyzed. Specifically, the associated costs to the execution and storage of the involved smart contracts are estimated, in order to determine the viability of the presented protocols. In addition, some possible improvements for the storage model, which could further reduce the associated costs, are also discussed. For last, the performance and timing of execution are evaluated in section 5.2.

5.1. Storage and Execution Costs Analysis

As stated in the objectives description, one of the main goals of this research was to create a simple proof-of-concept, without the use of complex and expensive homomorphic operations or zero knowledge proofs. For this reason, the only cryptographic operation implied is a standard hash function (SHA256), in addition to basic arithmetic checks and data storage in arrays.

Figure 8: Proof-of-life scheme smart contract source code

```
Initially deployed to Ethereum blockchain by an attacker $\mathcal{A}$.
Functions called by attacker
init(owner_address, max_free_files, min_ransom)
    · Set smart contract owner address to attacker's, owner_address := $\mathcal{A}$
    · Set the maximum number of files to be decrypted for free by an user to max_free_files.
    · Set minimum ransom amount to be paid to min_ransom.
    · Set array free_files_IDs := {}, containing the IDs of files that each victim has requested to decrypt as a proof of life.
getSamplesIDs()
    · Return free_files_IDs.
revealSamplesDecryptionKeys($id_v, \{id_f^*\}$)
    · Set victim.free_decryption_keys := $df_{f*}$
revealMasterDecryptionKey($id_v, msk_v$)
    · Assert victim.commitment $= h(msk_v)$
    · Set victim.state := REVEALED
    · Set victim.decryptionKey := $msk_v$
    · If $T < T_{end}$:
        · Set ledger[$\mathcal{A}$] := ledger[$\mathcal{A}$] + $amount

Functions called by victims
requestSamplesDecryption($id_v$, $\{id_f^*\}$, $c$)
    · Assert that len($\{id_f^*\}$) $\leq$ $max_free_files.
    · Set victim.free_files_IDs := $\{id_f^*\}$.
payRansom(amount, $id_v$)
    · Assert that $amount $\geq$ $min_ransom.
    · Add victim's ID to paid_IDs array.
getSamplesDecryptionKeys($id_v$)
    · Return victim.free_decryption_keys
getMasterDecryptionKey($id_v$)
    · Assert victim.state = REVEALED
    · If victim.state = PAID and T $> T_{end}$:
        · Set ledger[$\mathcal{V}$] := ledger[$\mathcal{V}$] + $amount
    · Return victim.decryptionKey
```

Figure 9: Proof-of-life scheme architecture

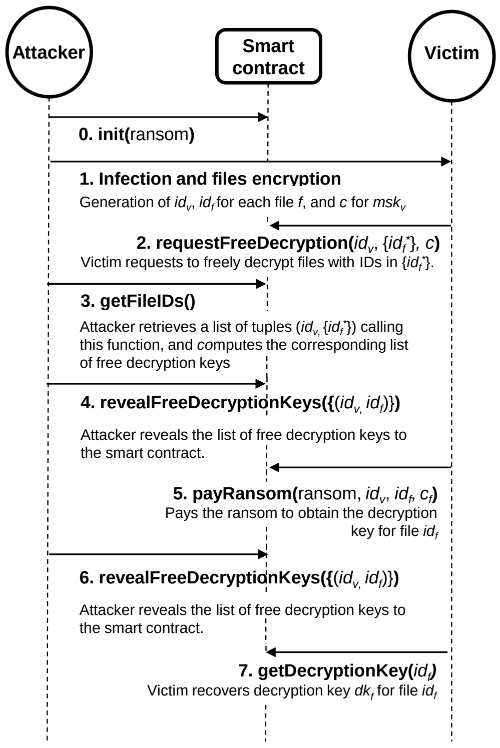

Attacker
Smart contract
Victim
0. init(ransom)
1. Infection and files encryption
Generation of id_v, id_f for each file f, and c for msk_v
2. requestFreeDecryption(id_v, {id_f^*}, c)
Victim requests to freely decrypt files with IDs in {id_f^*}.
3. getFileIDs()
Attacker retrieves a list of tuples (id_v, {id_f^*}) calling this function, and computes the corresponding list of free decryption keys
4. revealFreeDecryptionKeys({(id_v, id_f)})
Attacker reveals the list of free decryption keys to the smart contract.
5. payRansom(ransom, id_v, id_f, c_f)
Pays the ransom to obtain the decryption key for file id_f
6. revealFreeDecryptionKeys({(id_v, id_f)})
Attacker reveals the list of free decryption keys to the smart contract.
7. getDecryptionKey(id_f)
Victim recovers decryption key dk_f for file id_f

Table 2: Execution and non-volatile storage costs for the functions of the smart contract implementing the *pay-and-pray* scheme in Ethereum. Note that calls to read-only functions as *getVictims()* are free of charge (if not made by another smart contract). n is the number of victims. We have considered a gas price of 3 gwei (1 gwei $=10^{-9}$ETH), and 1 ETH =$170 (at time of writing, April 2019 [46]).

Step	**Party**	**Action / Smart contract call**	**Cost**		**Performance**
			Gas (EHT)	*USD*	*Average execution time (secs)*
0	Attacker	*init()*	791,412 (0.002374 ETH)	$0.4	19.19
1	Attacker	Infection and user files encryption	-	-	-
2	Victim	*payRansom()*	133,695 (0.000401 ETH)	$0.068	13.20
3	Attacker	*getVictims()*	-	-	0
4	Attacker	*revealDecryptionKey()*	69,170 (0.000208 ETH)	$0.035 (per victim)	11.37
5	Victim	*getDecryptionKey()*	22,735 (0.000068 ETH)	$0.011	14.40
	Total	**Attacker**	860,582 (0.002582 ETH)	0.4∗0.035n	38.97
		Victim	156,430 (0.000469 ETH)	$0.079	

The developed smart contract has been written in Solidity language, and deployed to the Ethereum Ropsten testnet[1], where it can be verified with any blockchain explorer like Etherscan [45]. It has been intentionally not optimized and its use has been restricted to specific users.

Table 2 shows the execution costs for each step (function) of the *pay-and-pray* protocol. The rest of protocols, *pay-per-decrypt* and *proof-of-life*, have similar costs (even slightly smaller) and are not shown for the sake of brevity.

Results show that *the costs of running the full protocol are clearly affordable* for all parties, overall considering the usual ransom amounts involved (in the order of 0.5 BTC or more).

For example, running a full campaign similar to WannaCry, with more than 300,000 affected users, would have an total cost of approximately $10,500 in execution and storage fees. However, this cost would not be paid by the attackers, but would probably be included in the ransom requested from the victims. In any case, the added cost respect to existing ransomware schemes is less than one cent of a dollar for each victim.

Specifically, the attack setup phase cost (smart contract deployment) is less than one dollar (step 0). After that, the ransomware encrypts the victims' files and they start paying the ransom (steps 1 and 2). Then, attackers receive an event with each payment (step 3), and the corresponding decryption key is calculated and revealed calling to the function *revealDecryptionKey()* (step 4). Finally, victims are signaled of the revelation with an event, which fires the call

[1]The smart contract has been deployed to the address `0x103193bff911d8ef979f8ba0015d36fbb3e72c17`.

Table 3: Non-volatile storage costs in Ethereum. We have considered a gas price of 1 gwei (1 gwei = 10^{-9} ETH), and 1 ETH = \$140 (at time of writing, March 2019).

Operation	**Gas/KB**	**ETH/KB**	**\$/KB**
READ	6,400	0.000032	\$0.004
WRITE	640,000	0.0032	\$0.448

to *getDecryptionKey()*.

As can be observed, some retrieval operations, such as *getVictims()*, have an associated storage cost and execution time of zero. This is because this function uses read-only operations and is therefore free of charge. In addition, it can also be considered immediate in terms of execution time, since the request is processed by the local Ethereum node, and does not reach the network.

The total execution costs for attackers would be $0.4 + 0.035n$ dollars, where n is the number of victims who have paid the ransom. For victims, the cost is negligible compared to ransom (and, in any case, included in it). On the other hand, it is well known that cryptocurrencies usually suffers sharp prices fluctuations. However, even if the value of the Ether (the Ethereum cryptocurrency) were to multiply its value several order of magnitude in the near future, the running costs of these protocols would still be affordable.

From this total cost, most of it is for storage of decryption keys during the revelation phase. In general terms, the storage space in public blockchains is specially expensive compared to computation, in order to discourage its abusive use. For example, the prices of non-volatile storage in Ethereum can be found in Table 3. For the sake of comparison, storing 1KB in a cloud service like AWS or Google Cloud costs about \$0.025 dollars per gigabyte, more than seven order of magnitude cheaper than Ethereum.

Although, as we have seen, the costs are perfectly affordable, attackers could use some techniques to reduce them even more. For example, by using the *transaction's log*, a special data structure in the blockchain. These logs were designed to use significantly less gas than usual contract storage. Specifically, logs basically cost 8 gas per byte, whereas contract storage costs 20,000 gas per 32 bytes.

On the other hand, the data is not accessible from within any smart contract (not even the contract that created it), but that is not a problem in our scenario. By using this technique, attackers could use these logs as a kind of public listing, in which each user could later retrieve their decryption key.

In summary, our experiments show that cost would not be a limiting factor and that, at least from this point of view, these schemes would be also viable.

5.2. Performance Analysis and Scalability Issues

Public blockchains are known to suffer from scalability problems. For example, it is estimated that Ethereum can process a few dozen transactions per second, although possible solutions are being actively worked on [47, 48].

In this section we analyse whether this limitation can have a significant impact on the proposed schemes. To this end, the execution times of the operations of each scheme have been measured, and the total time calculated. Times have been measured performing each operation ten times, discarding the minimum and maximum times, and calculating the average of the rest. It is important to note that the tests have been carried out in the Ropsten testnet, where the confirmation times are higher and have greater variability than in the mainnet.

Results shown in Table 2 demonstrate that proposals are also viable in terms of execution time and performance. As can be seen, the execution time is between 10 and 20 seconds for each operation, which would allow completing a full protocol round (payment/decryption key realese) in less than one minute. This speed of response, unlike the hours or days of traditional ransomware schemes, along with the automation capability, could represent a great incentive for the system to be adopted by attackers.

6. Countermeasures and Future Work

Unfortunately, the virtually immutable nature of public blockchains makes finding any kind of countermeasures to this threat extremely difficult. Indeed, smart contracts cannot be blocked, deactivated or removed if the author has not explicitly included appropriate mechanisms designed to do so.

Therefore, there are hardly any references in the literature about this possibility. Marino and Juels analyze in [49] the different possibilites for altering or undoing smart contracts, but always from a legal perspective, and with the contract owner permission (which obviously doesn't apply to our scenario).

As stated, unfortunately there exist very few realistic countermeasures, beyond causing an intentional hard-fork in the blockchain to eliminate the malicious smart contracts. However, in practice this would be extremely difficult, if not impossible. Similar previous situations, like the infamous DAO attack [50], seriously put the whole system at risk, when a hard-fork was forced to refund the theft of more than 3.6 million ETH, the equivalent of $70 million at the time. However, this decision was strongly contested by many Ethereum users, who argued that the hard-fork violated the basic tenets of blockchain technology.

For all these reasons, it seems clear that it is very unlikely that users not affected by ransomware attack based on the schemes described in this work were willing to repeat the history. As a result, this lack of effective countermeasures makes these schemes potentially very dangerous.

6.1. Future work

As previously mentioned, one of the main goals of this work is to demonstrate the viability of these schemes, so they have been deliberately designed as simple as possible, in order to keep complexity and execution costs at a minimum.

They could therefore greatly improved from various points of view, which could be the subject of future research. The following possibilities will be briefly analyzed below: (i) use of public-key cryptography, (ii) zero knowledge proofs and (iii) state channels.

Public-key cryptography and ZKPs. Existing ransomware schemes use public-key cryptography to send back encryption keys to the attackers. However, when trying to mimic this approach using smart contracts, a series of specific drawbacks arise: (i) prove that the attacker has the corresponding private key, and (ii) reveal it when necessary.

In principle, it might seem that both problems could be solved with the use of zero knowledge proofs, which allow to demonstrate that a file was encrypted with the private key corresponding to a given public key. Although these proofs exist, they wouldn't be very useful in the ransomware scenario. The reason is twofold. Firstly, the additional level of indirection (the file is actually encrypted with a symmetric key, encrypted in turn with a public key) would further complicate the design and implementation of the proof. Secondly, in any case, as outlined in section 3.3, this kind of proof does not help with the *semantic* verification of the property of the encrypted file, but only with the verification of the encryption keys.

State channels. On the other hand, it would be interesting to explore the possibilities that state channels could offer to this type of blockchain-assisted ransomware and to malware in general.

In addition, although the cost is not a limiting factor, these channels could help to reduce the exchanged messages and, therefore, the final cost even more. However, state channels require direct communication between parties, which could imply a great risk for attackers. A privacy layer could be added to address this issue.

7. Conclusions

In this work, we have shown how cybercriminals could benefit from the use of smart contracts running in public blockchains for carrying out ransomware attacks. We have foresee and presented several proposals for the semi-automation of ransomware schemes, relying only on symmetric cryptographic primitives and simple arithmetic operations.

These novel protocols have been implemented as a proof-of-concept in the Ethereum Ropsten testnet, with the aim of demonstrating its viability. In addition, the use of smart contracts enables the inclusion of new capabilities for ransomware, such as the *pay-per-file* paradigm, that would be difficult to implement in traditional schemes. To understand the potential risk of this threat, we have also evaluated the most important challenges and limitations an attacker would face in their implementation, along with their advantages and disadvantages.

In addition, we have analyzed the costs associated to the execution of these protocols in a real Ethereum environment. The results show that the costs are very low, which supports the idea of the viability of this threat also from a economic point of view.

Therefore, the conclusions are clear. This new kind of blockchain-assisted malware seems totally feasible, both from a technical and economical perspective. In fact, the authors believe that it is only a matter of time till they are found in the open, in the form presented here or in a similar variation.

Unfortunately, as stated before, there is currently no effective or realistic measure against such schemes. It is urgent, therefore, for the blockchain community to recognize and analyze this problem, in order to work on new and innovative solutions, which may counteract these threats.

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2009).
URL "http://bitcoin.org/bitcoin.pdf"

[2] M. Pilkington, Blockchain technology: principles and applications, in: Research Handbook on Digital Transformations, Edward Elgar Publishing, 2016. doi:https://doi.org/10.4337/9781784717766.00019.

[3] C. Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, 1st Edition, Apress, Berkely, CA, USA, 2017.

[4] A. Juels, A. E. Kosba, E. Shi, The ring of gyges: Investigating the future of criminal smart contracts, IACR Cryptology ePrint Archive 2016 (2016) 358.

[5] I. Eyal, A. E. Gencer, E. G. Sirer, R. V. Renesse, Bitcoin-ng: A scalable blockchain protocol, in: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), USENIX Association, Santa Clara, CA, 2016, pp. 45–59.
URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[6] A. L. Franzoni, C. Cárdenas, A. Almazan, Using blockchain to store teachers' certification in basic education in mexico, in: 2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT), Vol. 2161-377X, 2019, pp. 217–218. doi:10.1109/ICALT.2019.00070.

[7] J. Cheng, N. Lee, C. Chi, Y. Chen, Blockchain and smart contract for digital certificate, in: 2018 IEEE International Conference on Applied System Invention (ICASI), 2018, pp. 1046–1051. doi:10.1109/ICASI.2018.8394455.

[8] B. Notheisen, J. B. Cholewa, A. P. Shanmugam, Trading real-world assets on blockchain, Business & Information Systems Engineering 59 (6) (2017) 425–440. doi:10.1007/s12599-017-0499-8.
URL https://doi.org/10.1007/s12599-017-0499-8

[9] Y. Yuan, F. Wang, Towards blockchain-based intelligent transportation systems, in: 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), 2016, pp. 2663–2668. doi:10.1109/ITSC.2016.7795984.

[10] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, A. Gervais, Commit-chains: Secure, scalable off-chain payments, Cryptology ePrint Archive, Report 2018/642, https://eprint.iacr.org/2018/642 (2018).

[11] K. M. Khan, J. Arshad, M. M. Khan, Secure digital voting system based on blockchain technology, Int. J. Electron. Gov. Res. 14 (1) (2018) 53–62. doi:10.4018/IJEGR.2018010103.
URL https://doi.org/10.4018/IJEGR.2018010103

[12] R. Hanifatunnisa, B. Rahardjo, Blockchain based e-voting recording system design, in: 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA), 2017, pp. 1–6. doi:10.1109/TSSA.2017.8272896.

[13] D. Tse, B. Zhang, Y. Yang, C. Cheng, H. Mu, Blockchain application in food supply information security, in: 2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2017, pp. 1357–1361. doi:10.1109/IEEM.2017.8290114.

[14] H. M. Kim, M. Laskowski, Toward an ontology-driven blockchain design for supply-chain provenance, Intelligent Systems in Accounting, Finance and Management 25 (1) (2018) 18–27. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/isaf.1424, doi:10.1002/isaf.1424.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/isaf.1424

[15] H. Kaur, M. A. Alam, R. Jameel, A. K. Mourya, V. Chang, A proposed solution and future direction for blockchain-based heterogeneous medicare data in cloud environment, Journal of Medical Systems 42 (8) (2018) 156. doi:10.1007/s10916-018-1007-5.
URL https://doi.org/10.1007/s10916-018-1007-5

[16] A. Young, M. Yung, Cryptovirology: extortion-based security threats and countermeasures, in: Proceedings 1996 IEEE Symposium on Security and Privacy, 1996, pp. 129–140. doi:10.1109/SECPRI.1996.502676.

[17] R. Brewer, Ransomware attacks, Netw. Secur. 2016 (9) (2016) 5–9. doi:10.1016/S1353-4858(16)30086-1.
URL https://doi.org/10.1016/S1353-4858(16)30086-1

[18] E. Kalita, WannaCry Ransomware Attack: Protect Yourself from WannaCry Ransomware Cyber Risk and Cyber War, Independently published, 2017.

[19] J. Berr, 'WannaCry' Ransomware Attack Losses Could Reach $4 Billion [cited 07.01.2019].
URL cbsn.ws/2rluoXx

[20] J. Hernandez-Castro, E. Cartwright, A. Stepanova, Economic Analysis of Ransomware, SSRN Electronic Journal.

[21] M. Conti, A. Gangwal, S. Ruj, On the economic significance of ransomware campaigns: A bitcoin transactions perspective, CoRR abs/1804.01341.

[22] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, J. Hoffmann, Mobile-sandbox: Having a deeper look into android applications, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, New York, NY, USA, 2013, pp. 1808–1815. doi:10.1145/2480362.2480701.
URL http://doi.acm.org/10.1145/2480362.2480701

[23] F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Ransomware steals your phone. formal methods rescue it, in: 36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems - Volume 9688, Springer-Verlag, Berlin, Heidelberg, 2016, pp. 212–221. doi:10.1007/978-3-319-39570-8_14.
URL https://doi.org/10.1007/978-3-319-39570-8_14

[24] R. Shinde, P. V. der Veeken, S. V. Schooten, J. van den Berg, Ransomware: Studying transfer and mitigation, in: 2016 International Conference on Computing, Analytics and Security Trends (CAST), 2016, pp. 90–95. doi:10.1109/CAST.2016.7914946.

[25] N. Kshetri, J. Voas, Do crypto-currencies fuel ransomware?, IT Professional 19 (5) (2017) 11–15. doi:10.1109/MITP.2017.3680961.

[26] K. Liao, Z. Zhao, A. Doupe, G. J. Ahn, Behind closed doors: measurement and analysis of cryptolocker ransoms in bitcoin, in: 2016 APWG Symposium on Electronic Crime Research (eCrime), 2016, pp. 1–13. doi:10.1109/ECRIME.2016.7487938.

[27] D. Chaum, A. Fiat, M. Naor, Untraceable Electronic Cash, Springer New York, New York, NY, 1990, pp. 319–327. doi:10.1007/0-387-34799-2_25.
URL https://doi.org/10.1007/0-387-34799-2{_}25

[28] M. Jakobsson, A. Juels, Proofs of work and bread pudding protocols, in: Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security, CMS '99, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1999, pp. 258–272.
URL http://dl.acm.org/citation.cfm?id=647800.757199

[29] I. Bentov, R. Kumaresan, How to Use Bitcoin to Design Fair Protocols, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 421–439. doi:10.1007/978-3-662-44381-1_24.
URL https://doi.org/10.1007/978-3-662-44381-1{_}24

[30] R. Kumaresan, T. Moran, I. Bentov, How to use bitcoin to play decentralized poker, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, ACM, New York, NY, USA, 2015, pp. 195–206. doi:10.1145/2810103.2813712.
URL http://doi.acm.org/10.1145/2810103.2813712

[31] R. Kumaresan, I. Bentov, How to use bitcoin to incentivize correct computations, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, ACM, New York, NY, USA, 2014, pp. 30–41. doi:10.1145/2660267.2660380.
URL http://doi.acm.org/10.1145/2660267.2660380

[32] V. Buterin, Ethereum white paper: A next generation smart contract & decentralized application platform [cited 24.10.2017].
URL https://github.com/ethereum/wiki/wiki/White-Paper

[33] G. Wood, Ethereum: a secure decentralised generalised transaction ledger [cited 24.10.2017].
URL http://gavwood.com/Paper.pdf

[34] N. Christin, Traveling the silk road: A measurement analysis of a large anonymous online marketplace, in: Proceedings of the 22Nd International Conference on World Wide Web, WWW '13, ACM, New York, NY, USA, 2013, pp. 213–224. doi:10.1145/2488388.2488408.
URL http://doi.acm.org/10.1145/2488388.2488408

[35] S. T. Ali, D. Clarke, P. Mccorry, Bitcoin: Perils of an unregulated global p2p currency, in: Revised Selected Papers of the 23rd International Workshop on Security Protocols XXIII - Volume 9379, Springer-Verlag New York, Inc., New York, NY, USA, 2015, pp. 283–293. doi:10.1007/978-3-319-26096-9_29.
URL http://dx.doi.org/10.1007/978-3-319-26096-9_29

[36] A. Küpçü, A. Lysyanskaya, Usable optimistic fair exchange, Comput. Netw. 56 (1) (2012) 50–63. doi:10.1016/j.comnet.2011.08.005.
URL http://dx.doi.org/10.1016/j.comnet.2011.08.005

[37] M. Prince, Empty ddos threats: Meet the armada collective [cited 24.10.2017].
URL https://blog.cloudflare.com/empty-ddos-threats-meet-the-armada-collective/

[38] I. Ray, I. Ray, Fair exchange in e-commerce, SIGecom Exch. 3 (2) (2002) 9–17. doi:10.1145/844340.844345.
URL http://doi.acm.org/10.1145/844340.844345

[39] H. Pagnia, F. C. Gärtner, On the impossibility of fair exchange without a trusted third party, Tech. rep., TUD-BS-1999-02 (1999).

[40] G. Maxwell, The first successful zero-knowledge contingent payment [cited 24.10.2017].
URL https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/

[41] S. Bowe, Demostranting zero-knowledge contingent payments [cited 07.01.2019].
URL https://z.cash/blog/science-roundup.html

[42] S. Bowe, Pay-to-sudoku [cited 07.01.2019].
URL https://github.com/zcash-hackworks/pay-to-sudoku

[43] G. Fuchsbauer, Subversion-zero-knowledge SNARKs, Cryptology ePrint Archive, Report 2017/587, https://eprint.iacr.org/2017/587 (2017).

[44] B. Parno, J. Howell, C. Gentry, M. Raykova, Pinocchio: Nearly Practical Verifiable Computation, in: Proceedings of the IEEE Symposium on Security and Privacy, IEEE, 2013.
URL https://www.microsoft.com/en-us/research/publication/pinocchio-nearly-practical-verifiable-computation/

[45] Etherscan [cited 07.01.2019].
URL https://etherscan.io/

[46] Cryptocompare [cited 07.01.2019].
URL https://www.cryptocompare.com/coins/eth/overview/BTC

[47] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: Scaling byzantine agreements for cryptocurrencies, in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, ACM, New York, NY, USA, 2017, pp. 51–68. doi:10.1145/3132747.3132757.
URL http://doi.acm.org/10.1145/3132747.3132757

[48] V. B. Joseph Poon, Plasma : Scalable autonomous smart contracts, 2017.
URL https://plasma.io/plasma.pdf

[49] B. Marino, A. Juels, Setting standards for altering and undoing smart contracts, in: J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, D. Roman (Eds.), Rule Technologies. Research, Tools, and Applications, Springer International Publishing, Cham, 2016, pp. 151–166.

[50] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts sok, in: Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204, Springer-Verlag New York, Inc., New York, NY, USA, 2017, pp. 164–186. doi:10.1007/978-3-662-54455-6_8.
URL https://doi.org/10.1007/978-3-662-54455-6_8

8. Smart contracts code

NOTE FOR EDITOR AND REVIEWERS

We have doubts about whether or not to publish the source code, so we would like to know the opinion of editor and reviewers. In any case, we could make it available only on request.

8.1. Pay-and-pray

```
pragma solidity ^0.5.0;

contract PayAndPray {

    // Minimum ransom amount to pay
    uint public ransomAmount;
    address public attacker;
    enum State { Paid, KeyRevealed }
    // Maximum time for the attacker to reveal keys (in hours)
    uint dueTime = 24 * 3600;

    // This is a type for a single proposal.
    struct Victim {
        address payable victimAddress;  // Victim address
        bytes32 keyCommitment;  // Key commitment
        bytes32 decryptionKey;  // Decryption key for this victim
        State state;
        uint paymentDate;
    }

    mapping(uint256 => Victim) victimsMapping;
    uint256[] victimIDs;

    // Set the Attacker address
    constructor () public {
        attacker = msg.sender;
        ransomAmount = 0;
    }

    modifier onlyAttacker() {
        require(msg.sender == attacker,"Only attacker can call this function.");
        _;
    }

    event KeyRevealed(string message);

    //
    // Functions called by victims
    //
    function payRansom(uint256 victimID, bytes32 commitment) public payable {
```

```
    // Check that the paid amount is correct
    require(msg.value >= ransomAmount, "Ransom amount insufficient.");

    // Add a new victim to array
    victimsMapping[victimID] = Victim({
            victimAddress: msg.sender,
            keyCommitment: commitment,
            decryptionKey: "",
            state: State.Paid,
            paymentDate: now
    });
    victimIDs.push(victimID) -1;
}

function getDecryptionKey(uint256 victimID) public payable returns (bytes32){

    // Recover victim data
    Victim storage victim = victimsMapping[victimID];

    // Must the ransom be refunded to victim?
    if (victim.state == State.Paid && victim.paymentDate > now + dueTime)
        // Refund the victim
        //msg.sender.send(ransomAmount);
        victim.victimAddress.transfer(ransomAmount);

    if (victim.state == State.KeyRevealed)
        // Return decryption key to victim
        return victim.decryptionKey;

}

// This function returns all the victims request. Of course,
// it could be optimized by attackers to only return the list
// by chunks if this is very large
function getVictims()
    public view
    onlyAttacker
    returns (uint256[] memory) {

    // Look for victims with keys undisclosed
    return victimIDs;
}

function revealKey(uint256 victimID, bytes32 decryptionKey)
    public
    onlyAttacker
    returns (uint) {
    // Check both arrays have same size
    //require(victimIDs.length == decryptionKeys.length, "Both arrays lengths differ.");
```

```
        // Recover victim data
        Victim storage victim = victimsMapping[victimID];

        // Check the commitment of the key
        require(victim.keyCommitment == sha256(abi.encodePacked(decryptionKey)),
                            "Key commitment does not match");

        // Store the victim’s decryption key
        victim.decryptionKey = decryptionKey;
        // Mark the key as revealed
        victim.state = State.KeyRevealed;

        // Launch an event that could be capture from JavaScript or
        // an external oracle
        emit KeyRevealed("Key for user revealed");

    }

}
```