Can I Crash on Your Mainframe: Ransomware with Good Intentions

Adina Johnson Ezra Goss

November 2018

1 Introduction

Technology is a traditionally utilitarian domain area. Almost by nature, the programmer must first find a problem to solve before implementing a solution. From the automation of global infrastructure to the Roomba, purpose often predates presence for the machines in our lives. However, as the fields of artificial intelligence and natural language processing bring the possibility of a machine who can discuss and act freely into a foreseeable future, we must ask ourselves: will we see a generally intelligent machine as a tool, or as something entitled to purposeless, free existence?

To simulate the affect of interaction with technology devoid of utility, we present the *My Old Pal Ransomware* platform. *MOPR* enables a decentralized network of consenting users to unconsciously create and spread non-malicious malware. The medium of malware was chosen because in terms of contrasts, friendliness is as far from the intent of traditional malware as purposelessness is from traditional technology. Since the transmission of malware from host to host is also a traditionally decentralized process, and since recent developments in smart contract design in decentralized networks have made it more possible, we chose the blockchain as a technical structure for *MOPR* .

In the following sections we provide the preliminary knowledge necessary to understand the technical problem present in implementing and designing the *My Old Pal Ransomware* platform. This is a fluid document and is in its early stages so expect changes regularly.

2 Preliminaries

This section will outline the background information sufficient to keep this paper self contained. First we will introduce the concept of a *smart contract* and give a high-level explanation about a contract's lifecycle [2.1]. We will then highlight a specific virtual machine architecture that we think is ideal for the *MOPR* platform [2.1.3].

Since the *MOPR* platform's contract scripting language (CSL) [**??**] will allow interaction with the host's operating system (OS) through the *MOPR* client, the *MOPR* client will need to prevent the contract's access to potentially dangerous OS commands and permissions. We will provide the necessary understanding of these security concerns [2.2] and outline some standard safeguarding strategies specific to the macOS operating system [2.2.1].

2.1 Smart Contracts

Intuitive understanding of *smart contracts* – how they are programmed; how they are executed; how they are verified – will be prioritized over the technical details of implementation. For readers interested in a more technical or rigorous explanation of smart contract execution, please refer to [Wood, 2014] or [Kalodner et al., 2018] for different examples of virtual machine architectures used in *smart contract* execution. If that sentence makes no sense to you, read the next two subsections and return to those references after gaining a more qualitative understanding of smart contracts in blockchain systems.

Section 2.1.1 will provide a basic conceptual framework for *smart contract* logic and execution. Then, section 2.1.3 will highlight the distinctive traits of a specific virtual machine we recommend for *MOPR* contract execution. Reasons for recommending this virtual machine will be provided in section **??**.

2.1.1 Conceptual Model

We will begin by stating a terse, complete definition. We will then unpack this definition and contextualize its discrete parts.

Definition 1. *A smart contract is a verifiable, logical proof that can act as an agent in a decentralized network.*

Let's first unpack what we mean by the term *proof*. In Definition 1, by *proof* we mean an exhaustive reasoning about some *input*. Informally, we mean that – given some *input* – the *smart contract* is programmed to reason about the *input* in a way that is clear and complete. A reader of the *smart contract* should be able to trace every decision made by the *smart contract* in response to the *input*.

Proofs are often *logical* by nature. That is, the reasoning of a *proof* adheres to some framework of logic. In this way, a reader of the *smart contract* should be able to understand the reasoning of the contract if they understand the *logical* framework the *smart contract* adheres to. While these explanations are abstract, any programmer is familiar with the notion of a *logical proof*. The following program

```
if input == 'entering'
  print('Hello!')
else if input == 'exiting'
  print('Goodbye!')
else
  error('Oh no!')
```

Listing 1: *Logical Proof* A

follows all of the conditions of a *logical proof*. The "input" variable is the proof's *input*, and the logical statements "if", "else if", "else", belong to the logical framework used to reason about the *input*. We don't need to execute this program to know what it will do given some *input*. This is the human-friendly benefit of the *logical proof*.

Definition 1 also states the *proof* must be *verifiable*. By this we mean that – given a *logical proof* – a human or machine can verify the conclusion of the *logical proof* is correct given the *input*. If our *input* was 'entering' for *logical proof* A, then we as humans can verify that the program should output 'Hello!' by just looking at the code. A machine could also verify this by reading the *input* and *output* of the program and checking to see if the logic of the *proof* should result in the *output*. *Logical proof* A is both human and machine *verifiable*.

Legal contracts can also be *human-verifiable*, semi-*logical proofs*, if written correctly. Consider the following example

> The recipient (Bob) of the loan of $5,000 agrees to pay back the debt to the lender (Alice) within 2 months of the date of signing.

This contract could pass in a court of law. If the loan is repaid in the 2 month period, then the logic present in the contract is satisfied. However, the contract does not stipulate what happens if the debt is not repaid. This matter would be handled by a small claims court and that process might not be as logically sound. The purpose of this example is to highlight the completeness of the definition in Definition 1. A viable *smart contract* must handle all the logical possibilities present in reasoning about the *input*.

Finally, the *smart contract* must be able to act as an agent in the decentralized network it's apart of. This just means that the logic of the *smart contract* must be able to interact with the decentralized network in the same way that any human can interact with it. If a *smart contract* is designed to transfer

currency from one Bitcoin amount to another, it should be able to do so in the same manner as any human would.

As a final note about *smart contracts* , the *input* to a *smart contract* is often called a *message*.

2.1.2 Contract Execution

2.1.3 Arbitrum

2.2 Security Sandbox

2.2.1 macOS

[TODO: EZRA/ADINA?how are we safeguarding/sandboxing]

The shittyware will operate in a new user space with strict permissions.

we have specific modules that are verifiable

3 *MOPR*

This section will give an overview of the *MOPR* platform, in which individual smart contract programs are called "shittyware."

3.1 Overview of Shittyware and Examples

Think of shittyware like your friend that has a tendency to make annoying requests of you. These are tasks the user must go out of their way to perform for the shittyware, mildly inconveniencing them. Shittyware and their tasks will have a "success" state that can be validated by an oracle, as well as a fail state–typically, the main fail state is simply the shittyware being deleted before its task is fulfilled. If the user ignores the request or does not complete it to the shittyware's satisfaction, the shittyware will repeat the request, potentially becoming more insistent and intrusive.

- Your shittyware is new to the city and still hasn't figured out the public transportation system. It opens up your default mapping application and asks for the nearest bus stop. The contract knows what the 'correct answer' is, and if given the wrong answer, the shittyware will respond that the bus stop identified is too far and ask for a closer one. This shittyware requires the right answer to be satisfied.

```
load with "correct" nearest bus stop defined
open map application
prompt user for nearest bus stop
if input == saved answer:
    thank user
    rate user well
    exit
else if input != saved answer:
    reprompt user
else if user deletes shittyware:
    rate user poorly
```

Listing 2: *Pseudocode*

- Your shittyware is down on its luck and needs a place to crash for a bit. Can you open up your mainframe to it? The shittyware isn't really great as a houseguest–it eats up your memory and keeps popping up trying to make conversation with you. Early eviction is a failure, but if you can tolerate this shittyware's ways for a week or two, it'll eventually find a new user to mooch off of and will be satisfied with your generosity.

```
load with desired length of stay and array of potential actions
while desired length of stay less than time spent on computer:
    at random intervals launch a random potential action
    if deleted and desired length of stay less than time spent on computer:
        rate user negatively
        exit
rate user positively
exit
```

Listing 3: *Pseudocode*

- Your shittyware has an important email to send but unfortunately lacks the arms necessary to get the task done. Can you help a virus out? The shittyware will dictate the contents to you, requiring you to send the exact message to the right email address to be satisfied.

```
load with desired email text and address
open email
audibly dictate address and email text
if email received on address that matches desired text:
    rate user positively
    exit
else if user deletes shittyware:
    rate user poorly
else:
    reprompt user and redictate email
```

Listing 4: *Pseudocode*

3.2 User Interface

The program provides a local interface to perform actions, including:

- Checking your score on the leaderboard
- Using tokens to write new shittyware or edit a visitor currently on your computer

3.3 Scores and Tokens

Satisfying your shittyware has its rewards. When leaving your computer, the shittyware will rate you based on its satisfaction, based on conditions defined in its contract. At the base, satisfaction is obtained by completing the shittyware's core task, but the score can also be affected by length of time before the task was completed, how many times the shittyware needed to reprompt, etc. The shittyware's rating will be averaged into the user's total score, which will be displayed on a publicly available leaderboard. Ties on the leaderboard are broken by number of shittyware ratings.

Additionally, when you complete a shittyware's request, upon leaving your system it will reward you with tokens proportional to your current score. These tokens can be spent to write or edit shittyware, described in more detail

3.4 Shittyware Visitation

Shittyware visits happen semi-randomly; after one shittyware leaves your computer, there will be a break before another shittyware comes to visit. This will partly depend on how many potential shittyware there are available in the network. Only one shittyware can exist on the user's computer at a time.

By default, shittyware will not visit the same user twice. However, due to the restriction that a user can only edit the shittyware that is currently on their computer (see 3.5), a user has the option to invite a shittyware to return to their computer.

Shittyware contains friendliness ratings, codified by its author. A users rating (described in the section above) will determine what shittyware can visit them; for example, if a user has [TODO: just need to decide on specific scale]

3.5 Writing and Editing Shittyware

With the tokens described above, users will have the option to either create a new and exciting shittyware creature of their own or modify their current mainframe visitor. As described above, the system offers its own scripting language, CSL.

References

Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S. M., and Felten, E. W. (2018). Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1353–1370. USENIX Association.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32.