YaleNUSCollege

Synchronisation Primitives for Smart Contracts

Jake Goh Si Yuan

Capstone Final Report for BSc (Honours) in Mathematical, Computational and Statistical Sciences

Supervised by: Dr. Ilya Sergey

Acknowledgements

1. Associate Professor Ilya Sergey for his advice and wisdom as my main advisor, helping me to shape out this entire project from the wrinkle of an idea it was, and giving it space to bloom like the results that it is.

2. Dr Timothy M. Wertz for his guidance with proof and probabilistic analysis, and for having taught me those classes that gave me the mathematical foundations and insights to complete this project.

3. Professor Olivier Danvy for his assistance with affairs within and without academia, and constructive comments on how to improve on all aspects of this capstone endeavour.

YALE-NUS COLLEGE

Abstract

Mathematical, Computational and Statistical Sciences

B.Sc (Hons)

Synchronization Primitives for Smart Contracts

by Jake GOH Si Yuan

We explore the use of synchronization primitives in smart contracts, and make inferences about possible liveness side effects that could occur from the introduction of said primitives. From these possible side effects, we created several recovery and alleviation mechanisms that can be used as extensions with these synchronisation primitives, and prove that these mechanisms can provide the protection that is claimed. Finally, we perform some analysis with regards to a proof-on-concept analysis and discuss about the limitations and possible ways of improving on these limitations.

Contents

Acknowledgements **iii**

Abstract **iv**

1 Background and Motivations **1**

1.1 Background 1

1.1.1 Synchronization Primitives 2

1.1.2 Non-triviality of mutexes in smart contracts 3

2 Locking System **4**

2.1 Locking in Smart Contracts 4

2.1.1 Toy Example 4

2.2 The toy example and a simple naked mutex 5

2.3 The synchronized function modifier 6

2.3.1 Mechanism Overview 6

2.3.2 Lock Record 7

2.3.2.1 Generalised Lock Record for Single Variables 7

2.3.2.2 Generalised Lock Record for Single and Dynamic Variables 8

2.3.3 Synchronized modifier for dynamic variables . . . 9

2.3.3.1 Expanded Toy Example 9

2.3.3.2 Synchronized Dynamic 9

2.4 Liveness and Denial of Service 10

2.4.1 Denial of Service example 10

2.5 Recovery and Alleviation Mechanisms 11
2.5.1 Timeouts and sessions 11
2.5.1.1 Actor focused 12
2.5.1.2 Mechanism Overview 13
2.5.1.3 Limitation 13
2.5.2 Exponential deposits and decaying refunds 14
2.5.2.1 Fair usage through exponential deposits . 15
2.5.2.2 Decaying deposit refunds 15
2.5.2.3 Mechanism Overview 16
2.5.2.4 Limitation 17
2.5.3 Whitelisting and registration 18
2.5.3.1 Time limits 19
2.5.3.2 Registration fee 19
2.5.3.3 Refund holding period 20
2.5.3.4 Mechanism Overview 20
2.5.3.5 Limitation 22
2.5.4 Lock limits per session 22
2.6 Data Structures 22
2.6.1 Lock Record 22
2.6.2 User Record 23
2.6.2.1 Session Record 23
2.6.2.2 Deposit Record 23
2.6.2.3 Whitelist Record 24

3 Experiment and Analysis **25**
3.1 Implementation 25
3.1.1 Config Used 25
3.2 Analysis 26
3.2.1 Safety Property of Recovery and Alleviation Mechanisms 26

3.2.1.1 Optimal Strategy for Attacker 26
3.2.1.2 Assumptions and Simplifications Made . 27
3.2.1.3 Proof of Safety Property 27
3.2.2 Gas and Economic Costs 30
3.2.2.1 Explaining the differences 31

4 Discussion **33**
4.1 Usability Friction 33
4.2 Resource Costs 33
4.3 Liveness and Distributed Denial of Service 34

5 Related Work **35**
5.1 Adding concurrency to smart contracts 35

6 Conclusion **36**

Bibliography **37**

Dedicated to Golden Goh Kok Chye.

Chapter 1

Background and Motivations

1.1 Background

Blockchains are distributed databases that allow for write operations through append-only transactions. These transactions are evaluated and executed in sequential order. By that nature, concurrency problems are not possible as there would not be any overwriting over existing data. However, there have been new abstractions built on top of its distributed database essence that makes it possible for concurrent issues to arise. One such abstraction that we will focus on, is the smart contract, which is essentially a general purpose stateful computational object that lives on the blockchain.

Although the smart contract creation code and the method executing codes are appended to the blockchain via transactions that follow sequential ordered execution, two possible sources of concurrency problems arise:

1. The execution of a complete smart contract computation may take place over multiple transactions and use intermediary states, therefore leaving these states susceptible for concurrent access between the transactions.

2. Multiple smart contract methods, within the same contract and/or with other contracts, can be executed in a single transaction. There may be intermediary states shared by these methods whose integrity can be violated by reentrant access.

These sources of concurrency problems have arised in exploits, such as the multiple chances with one ticket in the Block King contract [1] which is an example of the first source of problem, and the infamous re-entrancy attack on the DAO [2] which is an example of the second source of problem. There is also a recognized class of problems classified as front-running, or transaction ordering dependence, which stems from the first source.

1.1.1 Synchronization Primitives

Despite the prevalance of concurrency problems in smart contracts, there has only been limited attempts done to introduce synchronization primitives. As of right now, there are no such primitives that exist on the virtual machine level. Instead, they are done on the application, or the smart contract, layer through programming language scripting. Most notably, there is the OpenZeppelin re-entrancy guard [6], which is a simple mutex that was designed and therefore limited to only solve concurrency problems within one transaction.

In this paper, we will explore the use of mutexes on the application layer for concurrency problems stemming from multi-transactions executions. By using account addresses as the unique identifier, and some state in the smart contract, we are able to build out a mutex that can be used across both single and multiple transaction(s) executions.

1.1.2 Non-triviality of mutexes in smart contracts

The use of mutexes in smart contracts, especially over multiple transactions, is a non-trivial problem as the common mutex side effects of deadlock and starvation can easily occur and may cripple the smart contract's functionality. Smart contract method executions can only be triggered via manual transactions and not automated within the blockchain, the smart contract may be permanently disabled without carefully designed recovery mechanisms. The decentralized nature of the blockchain suggests that such mechanisms have to be carefully built along with the locking logic in order to not create situations where the smart contracts are stuck in lock. For instance, a standard user may take up the lock for a smart contract with only one state variable, and have an incentive to keep it continually locked, with the intention to deny service to any other user. Using standard mechanisms such as timeouts or queues directly may not be effective as it is relatively easy to circumvent that through daisy chaining lock-release and using multiple addresses. The relative ease of using multiple addresses in particular, spins up some complications which are closer in form to distributed denial of service attacks.

At the same time, computations and access to memory and state costs actual value, such as in the **gas** model [4] in the Ethereum blockchain. Therefore, any *useful* design of mutexes in smart contracts has to minimize its resource usage, whilst maintaining its concurrency security guarantees.

Chapter 2

Locking System

2.1 Locking in Smart Contracts

Problems with concurrent use of shared resources carry a similar form. There is an arbitrary piece of persistent data that an actor, be it a thread or an account, relies on for some computation. This computation usually requires multiple access over different steps, be it read or write, to the data. The integrity of the computation relies on the data being unchanged unless explicitly intended by the actor. In other words, these problems occur when other actors modify or write to the data before the computation of the original actor is complete. The use of mutexes, or locks, can solve this problem as it will only allow permissioned access to the shared data to the original actor, and protect it from concurrent access that may disturb the integrity of the computation.

2.1.1 Toy Example

Before I begin with describing the solution, I will first describe a base example of a contract that may be fallible to concurrency issues.

```
1 contract Example {
2   uint foo;
3
4   function increaseFoo(){
5     foo = foo + 1;
```

```
6   }
7
8 }
```

In this example, there is a state variable **foo** that can be increased by calling the function **increaseFoo**. We can quickly see how that may lead to a concurrency issue with the following scenario with two actors **Alice** and **Bob**.

Let us assume that **foo** starts from 0, and **Alice** would like to make **foo** have the value of 2. In the first block, only **Alice** calls the function and increases **foo** to 1. In the next block, **Alice** calls the function again with the intention and expectation that **foo** would go to 2. However, before **Alice**'s transaction is executed, **Bob** calls the function, increasing **foo** to 2. **Alice**'s transaction executes after, bringing **foo** to a final value of 3 instead of 2 as expected.

2.2 The toy example and a simple naked mutex

Let us now solve the toy example with the scenario with the lock design. The solution takes a leaf out of Java's book[5], by introducing a **synchronized** function modifier.

```
1 contract Example {
2   uint foo;
3
4   function increaseFoo() synchronized("foo"){
5     foo = foo + 1;
6   }
7 }
```

In the solution, the **synchronized** function modifier takes in a parameter that references the shared state that would be modified in the function. In this case, the name of the shared state is referenced. The **synchronized**

code block is executed before the main **increaseFoo** code block. The **synchronized** code block encapsulates all the different mechanisms that has to be run to ensure the proper locking of **foo**. I will elaborate on this in a later section.

Let us go back to the scenario. **Alice** first runs **increaseFoo**, bringing it to 1, and at the same time taking the lock. Again, **Bob** tries to call the function before **Alice** is able to do it the second time. However, as **Bob** does not currently hold the lock, his transaction is rejected. **Alice**, being the holder of the lock, or locker, is able to call the function the second time, therefore bringing **foo** to 2, as she intended to do so in the first place.

2.3 The synchronized function modifier

Here, I will go into a more in depth explanation about how the solution works, through looking at the **synchronized** function modifier

2.3.1 Mechanism Overview

When the **synchronized** code block is triggered, using the toy example, here is what happens

1. Check if **foo** is locked, if it is move to step 2, else move to step 3.
2. Check if the current locker of **foo** is the actor, if so, proceed to step 4. Else, terminate and return.
3. Lock **foo** and set the locker to the actor, proceed to step 4.
4. Continue to the main function normally

2.3.2 Lock Record

We can see that step 1 and 2 implies the existence of some persistent data that records whether a state variable, such as **foo** is locked, and who it is locked to. In the case of the toy example where there is only a single state variable, we would be able to use something like :

```
1 address fooLock;
```

If **fooLock** is set to the zero address, then **foo** is considered to be unlocked. However, if **fooLock** is set to a non-zero address, such as **Alice**'s address, then it is considered to be locked, with the locker being Alice. This way of representing the lock allows us to combine the lock status and the reference to the locker in one elegant sweep.

2.3.2.1 Generalised Lock Record for Single Variables

However, this only targets a single state variable. We can imagine that in a smart contract there might be multiple state variables that require concurrency protection. There is a need to generalize the specific solution above. This can be achieved by extending the lock record using a hash table like data structure, such as Solidity's **mapping**, so that we can use a reference to the state variable as the key, and the locker's address as the value.

```
1 mapping(bytes32 => address) lockMap;
```

I have chosen to use a state variable reference of type **bytes32** as using unbounded types, such as **string**, for mapping keys is not available as of the time of writing. However, it is not difficult to convert something of type **string**, such as the name of the state variable(if chosen as the reference), to **bytes32**, by using a fixed-length output hashing function such as **keccak256**.

2.3.2.2 Generalised Lock Record for Single and Dynamic Variables

Some smart contract may need fine grained access inside dynamic variables. By *dynamic* variables, I refer an array, a hashtable(like mapping), or an arbitrary struct. The current design makes it difficult, without some creativity in the part of the developer, to have such fine grained control and protect elements within dynamic variables without locking the whole thing up. We can again extend the current lock record design, by encapsulating it within another hash table.

```
1 mapping(bytes32 => mapping(bytes32 => address)) lockMap;
```

The key to the outer hash table would be the reference to the state variable. This state variable could be a single variable such as an **int** or **string**, or it could also be a dynamic variable. The value of the outer hash table would be another hash table. For a dynamic variable, the key to the inner hash table would be the same key used to retrieve the value in the dynamic variable. For a single variable, the key used would be a zero. The value for both cases would be the address of the locker.

For example, let's say there is a dynamic variable of **mapping** type of name **foo** and **Alice** would like to lock the data element which is referenced by a **uint** key of 123.

```
1 mapping(uint => string) foo;
```

To lock this particular data element, **Alice** would have to set **lockMap**[hash("foo")][hash(123)] to her own address. If **foo** were a single variable, then **Alice** would have to set **lockMap**[hash("foo")][hash(0)] instead. There are no risks of collisions as each state variables, single or dynamic, would have unique names.

2.3.3 Synchronized modifier for dynamic variables

With the introduction of dynamic variables, the **synchronized** function modifier has to be able to include the new data regarding key of dynamic variable.

2.3.3.1 Expanded Toy Example

Before we do that, let us expand the toy example to include a dynamic variable.

```
1  contract Example {
2    uint foo;
3    mapping(uint => uint) bar;
4
5    function increaseFoo() synchronized("foo"){
6      foo = foo + 1;
7    }
8
9    function increaseBar(uint key) {
10     bar[key] = bar[key] + 1;
11   }
12
13 }
```

2.3.3.2 Synchronized Dynamic

To protect an element inside of **bar**, a new function modifier named **synchronizedD**, shortened from **synchronizedDynamic** is introduced. The **synchronizedD** modifier has the same operational semantics as **synchronized** as described in 2.3.1. The difference between the two modifiers is the expanded parameters input to take in the key of the element as well. Therefore, with the toy example, the function would look like this :

```
1  function increaseBar(uint key) synchronizedD("bar", key){
2    bar[key] = bar[key] + 1;
```

```
3 }
```

Of course, it would be possible to re-use the same **synchronized** word and have it expanded. This would mean that for single state variables, the developer would just have to put a dummy zero input for the key parameter. However, this may lead to some confusion, and as a design choice, I have chosen to define it with a different name.

2.4 Liveness and Denial of Service

The astute reader who is familiar with distributed systems and/or smart contracts would notice that using the lock system in its most simple form would very easily lead to situations where deadlocks and starvation may occur. At the same time, if these situations are triggered by a malicious actor with the intention of slowing down or stopping the normal functionalities of the smart contract, then we can define it as a denial of service attack.

Unfortunately, as with the canonical denial of service attacks, there are no ways to completely stop this from happening without severely hampering the regular functionality of the subject of the attacks. However, there are methods that we can borrow from solutions to starvation and denial of service, combined with some cryptoeconomics, to reduce the impact of the attacks whilst not unfairly punishing regular usage.

2.4.1 Denial of Service example

Before we move on to introduce the solutions that I have come up with, I will illustrate with the single state variable toy example how a denial of service attack may occur. It will also apply for dynamic state variable

locks as there is no difference between the operational semantics of the two locks.

```
contract Example {
  uint foo;
  function increaseFoo() synchronized("foo"){
    foo = foo + 1;
  }
  function release(address locker){
    if(locker == msg.sender){
      unlock(locker);
    }
  }
}
```

In our discussions above, it was implied that there was a **release** function which can be used to unlock the locks held by a particular locker. In this example, this is made explicit to make the scenario clearer.

In normal usage, **Alice** would call **increaseFoo**, take the lock through **synchronized**, and continue with using it via **increaseFoo** or other functions that would also use the **foo** lock. When she is done, she would call **release** with her address and unlock **foo** for another actor to use. However, if **Alice** does not call **release**, whether out of forgetfulness or malice, then the lock would be permanently in place and she would have essentially rendered part of or all of the smart contract useless.

2.5 Recovery and Alleviation Mechanisms

2.5.1 Timeouts and sessions

To reiterate, there needs be a recovery mechanism for locks to be automatically released after the user is done, or has exceeded a certain fair

amount of usage time, with the lock. To introduce such a recovery mechanism, we can borrow a tried and tested mechanism from the standard concurrency toolbox– timeouts. This means that locks will in action only for a reasonable amount of time, where the state variable being locked will be exclusive to the actor. After the time has passed, the state variable is essentially unlocked and can now be locked and used by any actor.

In order for this to work, there has to be a timed property existing in persistent data somewhere, such that when a lock is first taken, the timed property can be set to either when the lock is taken, or when the lock is expired.

2.5.1.1 Actor focused

At the same time, there has to be some kind of reference between this timed property with either lock itself, or the actor who takes the lock. The typical implementation in the standard concurrency toolbox is to attach the timeout to the lock. This would mean that an actor would be able to take multiple locks concurrently. If we were to assume that there are malicious actors whose intent is to cause a denial of service attack, this choice would be suboptimal.

Therefore, it would be better to pick the latter and have timeouts that are attached to an actor, creating a mechanism that is more session-like. This means that when an actor will have a limited amount of time where he will be able to take locks and have exclusive access to the state variables associated with the locks. The amount of time is determined by when the first lock is taken, and the standard timeout duration stipulated by the smart contract. At the end of timeout, or if the actor calls **release** before then, all the locks held by the actor will be unlocked and available for use.

2.5.1.2 Mechanism Overview

To see how timeouts will work in steps, I will extend the **synchronized** mechanism and highlight the steps relevant to timeouts.

1. Check if **foo** is locked, if it is move to step 2, else move to step 4.
2. Check if the session of current locker of **foo** has expired. If so, move to step 4, else move to step 3
3. Check if the current locker of **foo** is the actor, if so, proceed to step 7. Else, terminate and return.
4. Check if actor has an existing session. If so, proceed to step 6. Else, move to step 5.
5. Set a new timeout for the actor. Proceed to step 6
6. Lock **foo** and set the locker to the actor, proceed to step 7.
7. Continue to the main function normally

For **release**, or the unlocking of all held locks, it will be a trivial loop that sets all locks held by the actor back to the zero address. Before timeouts, **release** was to be triggered exclusively by the locker. This exclusivity will still remain intact whilst the session has not timed out yet. However, when it has timed out, any actor would be able to call **release**.

2.5.1.3 Limitation

Timeouts help to introduce a recovery mechanism to the liveness issue. This solution would be enough if we were dealing with non-malicious actors only. However, that is an assumption that is naive, especially when one considers the events that have taken place in the short history of blockchain and smart contracts.

A malicious actor may exploit the fact that locking and unlocking is almost *free*. This means that the actor may lock and unlock simultaneously, and the state variable that is associated would for all functional purposes be held in a stasis for as long as the actor intends. Of course, this is not entirely *free*, as the malicious actor would still have to pay **gas** fees for every lock and unlock transaction that he executes. This idea does motivate us to think economically. If we were to assume rational actors in the system, then there would be ways to disincentivize and economically limit the malicious ones, as I will display in the next two sections.

2.5.2 Exponential deposits and decaying refunds

One of the simplest ways to introduce economic motivations into the system is to have the actor place a deposit with every lock that he takes. When the lock is returned by the locker, he will get a refund of the deposit that he placed. When used by itself, this will economically incentivize an actor to release the lock.

When used together with a timeouts and sessions system, the lock can be released by any other actor when the session has expired. The deposit will also go to the actor that triggers the release. This will form an economic incentive for other actors to "clean" the smart contract, thus increasing the total liveness of it.

However, in this simple form, it does not increase the cost that a malicious actor will incur in a strategy of simultaneous lock-unlock. The malicious actor will receive the full deposit back with each unlock that occurs, and just has to be careful to not let the session expire to prevent a third party unlocking it.

2.5.2.1 Fair usage through exponential deposits

To harden this mechanism and also encourage fairer usage, I introduce the concept of a **cool-off** and an exponentially growing deposit for repeated use. The **cool-off** is defined as a period of time, after a session has ended, whether released or expired, when the deposit for another session would be exponentially more expensive.

$$Deposit_{n+1} = Deposit_n * 2^{Time_{\text{end}} + Time_{\text{cooloff}} - Time_{\text{now}}}$$

This extension will be able to heavily disincentize an actor from having multiple successive sessions, therefore making the system much fairer by giving all other actors a better chance at taking a lock.

It will also alleviate the denial of service attempts through the lock-unlock strategy by a malicious actor, who now has a $Time_{\text{cooloff}}$ buffer period where he would not be able to perform the attack unless he is willing or able to pay a much higher deposit cost. In that $Time_{\text{cooloff}}$ period, other actors would be able to take the lock of state variables that were previously held by the the malicious actor, therefore reducing the effect of the denial of service.

2.5.2.2 Decaying deposit refunds

Another extension that could be use to harden this mechanism is through decaying deposit refunds. This means that the amount of deposit refunded to the actor is adjusted by how long he has used the session.

$$Refund = Deposit * \frac{Time_{\text{end}} - Time_{\text{now}}}{Time_{\text{standard timeout}}}$$

The current design, as can be seen from the equation above, sets the decay rate at a linear scale, which can be changed easily based on the requirements of the smart contract. For instance, the smart contract developer may feel that having the decay equation set like this may be unfairly punishing regular users of the smart contract. It would be trivial to adjust it such that decay only kicks in after a certain reasonable amount of time, so regular users would be able to receive their full deposits.

This extension will be able to incentivize an actor to promptly complete their computation and release the locks, so as to not lose too much of their deposit and incur unnecessary cost.

For the malicious actor with the lock-unlock strategy, this extension will force him to choose between increasing the frequency in which lock-unlock occurs so as to reduce the loss of deposit thus incurring more costs in transaction gas fees, and losing more of the deposit. In both cases, this extension will increase the cost of the malicious attack. When combined with exponential deposits, the malicious actor will be forced to balance between losing the deposit and shortening the attack time, providing a good amount of alleviation to the attack's effect.

2.5.2.3 Mechanism Overview

To see how deposit will work in steps, I will extend the **synchronized** mechanism with timeouts, and highlight the steps relevant to deposit.

1. Check if **foo** is locked, if it is move to step 2, else move to step 4.
2. Check if the session of current locker of **foo** has expired. If so, move to step 4, else move to step 3
3. Check if the current locker of **foo** is the actor, if so, proceed to step 8. Else, terminate and return.

4. Check if actor has an existing session. If so, proceed to step 6. Else, move to step 5.
5. Check if the actor has a sufficient amount of deposit. If so, proceed to step 6. Else, terminate and return.
6. Set a new timeout for the actor. Proceed to step 7
7. Lock **foo** and set the locker to the actor, proceed to step 8.
8. Continue to the main function normally

It is important to note that the exponential deposit and refund decay extensions do not affect the steps in the mechanism as they only change the conditions and effects of deposit.

For **release**, there is no difference in operational semantics. There is only one additional step after all the unlocking is done, for the **actor** who released the locks, be it the original locker or some other actor after the expiration of the locker's session, to be refunded the remaining deposit. Whether the deposit is done in a push or pull manner can be determined by the developer, although it should be noted that doing it as a push may possibly incur a re-entracy attack, like the DAO exploit, in a poor implementation.

2.5.2.4 Limitation

Having a deposit mechanism increases the amount of friction in the smart contract for a regular user. If the hardening extensions such as exponential deposits and decaying refunds are used, it would mean that a regular user may end up spending much more transaction costs than they would have without the deposit mechanism, and there would be a higher average latency. It must be noted that in a system with limited resources, any mechanism to encourage fair usage would lead to some additional

friction in utilization, and therefore this is something that is expected.

At first glance, a malicious actor implementing the lock-unlock strategy for a denial of service attack seems to be limited. This would be true if the actor is only limited to one uniquely identifying address that he can perform this attack from. However, that is clearly not the case in current smart contract systems. A malicious actor is able to coordinate and perform the attack from multiple addresses trivially, as there is no additional cost to having and using many addresses, thus making this a distributed denial of service attack. With this distributed lock-unlock strategy, the attacker can circumvent the exponential deposit extension by performing lock-unlock with different addresses successively, rotating with addresses with cool-off that has expired.

2.5.3 Whitelisting and registration

The distributed denial of service attack is viable when the cost of having multiple identities or addresses is relatively cheap, therefore making that action more expensive would help to alleviate such an attack. At the same time, we have to continue to balance punishing or preventing attackers with not adding too much friction or costs to regular users.

A mechanism used to alleviate standard distributed denial of service attack is whitelisting. This means that only certain identities are allowed to access the service being attacked. In our case, actors would have to register their addresses with the smart contract before they are allowed to interact with it.

2.5.3.1 Time limits

Whitelisting can be made more effective effective if there was a fixed time limit to how long the whitelist would be. Having an expiring registration for the whitelist would provide an additional step to the functional and resource cost of the malicious actor, who would have to continue registering after expiry to persist an attack.

This would not unnecessarily punish a regular user, as long as the whitelist period is set at a reasonable time that is able to cover the time it takes for regular use. The time it takes to carry out an effective distributed denial of service attack would typically be much longer than a regular use time.

2.5.3.2 Registration fee

If registration for whitelisting was free, then the whitelist mechanism would have done little to alleviate distributed denial of service attacks besides. Therefore, a refundable registration fee can be implemented to increase that cost. Having a registration fee would directly limit the number of addresses that a malicious actor can use at one time, as the actor would have a finite balance of tokens.

$$\text{Upper limit of Addresses} \approx \frac{\text{Total balance of Actor}}{\text{Registration fee}}$$

Note that this is only an approximation as there are also costs to executing the attack from transaction costs and deposits that has to be factored in as well. However, these costs should be much lesser than the registration fee so as to not unfairly punish regular users, so the biggest factor in determining the upper limit of addresses would be the registration fee.

2.5.3.3 Refund holding period

A refund holding period can be used to harden the registration fee and time limit extensions. The refund holding period can be defined as a period of time, after the whitelist period has ended, when the registration fee cannot be refunded. The registration fee can only be taken back after the refund holding period has passed.

By having a refund holding period, a malicious actor with a limited amount of resources would have even less addresses that he is able to utilize if he wants to perform an attack. The actor would have to choose between having entire periods of time equivalent to the refund holding period where there is no attack going on, or staggering the registration of different addresses at different periods of time. Choosing the former would lead to a suboptimal attack strategy, and choosing the latter would mean that there would be less addresses available to use. The smart contract developer may also be able to choose a registration fee and refund holding period such that the attacker may be limited to one address.

2.5.3.4 Mechanism Overview

To see how whitelisting will work in steps, I will extend the **synchronized** mechanism with whitelisting, and highlight the steps relevant to whitelisting.

1. Check if the actor has a non-expiring registration, if it is, move to step 2. Else, terminate and return.
2. Check if **foo** is locked, if it is move to step 3, else move to step 5.
3. Check if the session of current locker of **foo** has expired. If so, move to step 5, else move to step 4

4. Check if the current locker of **foo** is the actor, if so, proceed to step 9. Else, terminate and return.

5. Check if actor has an existing session. If so, proceed to step 7. Else, move to step 6.

6. Check if the actor has a sufficient amount of deposit. If so, proceed to step 7. Else, terminate and return.

7. Set a new timeout for the actor. Proceed to step 8

8. Lock **foo** and set the locker to the actor, proceed to step 9.

9. Continue to the main function normally

Whitelisting would also require components to register and to request for the refund of registration fee. It is relatively trivial, but for the sake of clarity I will go through the mechanisms.

For registration:

1. Check if actor has a sufficient amount of registration fee. If so, proceed to step 2. Else, terminate and return.

2. Update the actor's user record with registration with registration time and registration amount.

For refund of registration fee:

1. Check if actor's refund holding period has passed. If so, proceed to step 2. Else, terminate and return.

2. Update the actor's user record with registration refund and refund the amount to actor's address.

2.5.3.5 Limitation

The obvious limitation with whitelisting is the significant friction that it brings to the smart contract. There is now an additional transaction that an actor would have to send before being able to even use the contract. Since a non-trivial amount of registration fee is necessary for whitelisting to be effective, a regular user of the smart contract would need to have that amount of tokens before he is able to use it.

2.5.4 Lock limits per session

The smart contract that undergoes a denial of service attack has an attack surface of size determined by the number of state variables that can be locked. One of the most direct and simplest ways to alleviate the effect of the denial of service attack is to limit that attack surface. To do that, we can have a hard limit on how many locks a malicious actor is able to take in one session.

As long as the lock limit is set to the maximum number of locks needed for the longest, most complex computation in the smart contract, a regular user will not be hampered by this mechanism.

2.6 Data Structures

In this section I will describe the data structures that I have designed and implemented for the features to work.

2.6.1 Lock Record

As I have described above at 2.3.2

2.6.2 User Record

For most of the recovery and alleviation mechanisms to work, a record of the actor's actions is necessary. For User Record, I will break it down to the different components that it is composed of.

2.6.2.1 Session Record

The session record consists of an array to keep track of the locks that the actor has taken up in his session, so that unlocks and release can be performed more efficiently; and it also contains the expiry time of the session, and the time when all locks were released. The implementation may not have the different components as its own structs so as to make the library more gas-efficient.

```
struct LockInfo{
  bytes32 dataName;
  bytes32 key;
}
SessionRecord struct {
  LockInfo[] lockInfos;
  uint256 expiryTime;
  uint256 unlockTime
}
```

2.6.2.2 Deposit Record

The deposit record consists the amount of deposit paid for the current or last session, a boolean to reflect whether the deposit has been refunded.

```
DepositRecord struct {
  uint256 depositFee;
  bool depositRefunded;
}
```

2.6.2.3 Whitelist Record

The whitelist record consists of the time when the whitelist registration expires, and a boolean to reflect whether the registration fee has been refunded. The decision to not include the refund holding period is made so as to be more gas efficient. It is trivial to derive the refund holding period using an offset in the library config.

```
1 WhitelistRecord struct {
2   uint256 whitelistExpiry;
3   bool registrationRefunded;
4 }
```

Chapter 3

Experiment and Analysis

3.1 Implementation

The protocol and toy example described in the last chapter was implemented in Solidity, and can be found at https://github.com/jakegsy/synchronizationPrimitivesSmartContract/tree/master/contracts. The implemented smart contract is deployed on the Kovan Ethereum Testnet at the address 0xda2523d8eee5d096baed310bbafe0e48849da80.

3.1.1 Config Used

Session Period	5 Blocks
Cool Off Period	10 Blocks
Whitelist Period	45 Blocks
Refund Holding Period	90 Blocks
Deposit Fee	0.001 Ether
Whitelist Fee	0.1 Ether
Maximum Locks	3

3.2 Analysis

3.2.1 Safety Property of Recovery and Alleviation Mechanisms

In this section, I will use the recovery and alleviation mechanisms to show that given the right config, a distributed denial of service attack can be made to be unsuccessful. A successful distributed denial of service attack can be defined as one where all possible state variables are locked up by the malicious actor for as long he desires and has the balance to do so. To make things simpler, I will only use and try to show for the harder scenario of one state variable in the smart contract. It would be easy to see how the same safety property will not only be present, but also stronger in the existence of many state variables. I will use the toy example with only **foo** as the state variable.

3.2.1.1 Optimal Strategy for Attacker

The attacker performing a distributed denial of service attack has to ensure that he has all the unexpiring locks to all possible state variables for as long as possible. At the same time, the attacker wants to minimize the fees spent to execute this attack, though that is a secondary concern.

Therefore, the attacker would employ the strategy of locking and unlocking with the minimum amount of addresses. This way, the attacker would not have to be penalized for consecutive sessions, while maintaining ownership of the lock. The addresses would also be staggered in whitelist registration so as to minimize the waste of precious whitelist periods.

3.2.1.2 Assumptions and Simplifications Made

In this scenario, I assume that the attacker has only a finite amount of cryptocurrency that he is able to use in this attack. I will define the available balance to the attacker as **X**.

At the same time, I will also assume that all recovery and alleviation mechanisms are used in the smart contract, and it has a config of

$$\begin{aligned} \text{Session Period} &= a \\ \text{Cool-off Period} &= b \\ \text{Whitelist Period} &= c \\ \text{Refund Holding Period} &= d \\ \text{Registration Fee} &= Y \end{aligned}$$

Finally, I will assume that the attacker's transactions are always taken first and succesful, such that the transition of locks between addresses are always uninterruptible.

3.2.1.3 Proof of Safety Property

Lemma 3.2.1. *The maximum amount of whitelisted addresses that an actor with a balance of* ***X****, on a smart contract that has a whitelist fee of* ***Y****, can have at any point of time, is* $\lfloor \frac{X}{Y} \rfloor$.

Proof. Since the actor only has a total balance of **X**, discounting possible transaction fees, he is only able to register for $\frac{X}{Y}$ addresses at once. Since we do not know whether X is divisible by Y, we take $\lfloor \frac{X}{Y} \rfloor$ as there is no meaning for a partially registered address.

□

Lemma 3.2.2. *The maximum amount of sessions that an address, that is owned by an attacker who is using the optimal strategy, can have in a whitelist period of c is* $\lceil \frac{c}{a+b} \rceil$.

Proof. In the optimal strategy, the attacker would want to avoid using the address to have a session while it is in the cool-off period. Therefore, after every session period of **a**, the attacker would wait for the cool-off period of **b** to pass before using the address to take another session. As such, we can use **a+b** as the total time for a session, and derive the maximum amount from the whitelist period of **c**. □

Corollary 3.2.2.1. *The maximum amount of sessions that an address, that is owned by an attacker who is using the optimal strategy, can have in a whitelist period of c and refund holding period of d is* $\lceil \frac{c}{a+b} \rceil$.

Proof. We know from 3.2.2 in a period of c, the maximum number of sessions that an address can have is $\lceil \frac{c}{a+b} \rceil$. Since c is contained within $c + d$, and the particular address cannot take up additional sessions within d, we can expand the 3.2.2 to $c + d$. □

Lemma 3.2.3. *The minimum amount of sessions that an attacker must have in a* $c + d$ *period of time, to carry out the optimal strategy, is* $\lceil \frac{c+d}{b} \rceil$

Proof. Given the assumption that the transition of locks between sessions of different addresses is always uninterruptible, it is easy to see that for any arbitrary period of time, there must be sessions covering every moment for the optimal strategy to be carried out. Therefore, for a $c + d$ period of time, it has to be covered by $\frac{c+d}{b}$ successive sessions. □

Lemma 3.2.4. *The minimum amount of addresses that an attacker has to use, to carry out the optimal strategy, is* $\left\lceil \frac{\frac{c+d}{b}}{\lceil \frac{c}{a+b} \rceil} \right\rceil$, *where a is the session period, b is the cool-off period, c is the whitelist period and d is the refund holding period.*

Proof. First, let us consider the period of $c + d$. This can be considered one cycle of an address, where it is able to take up sessions in c and be idle in

d. The cycle of one address is important in an analysis of the minimum amount of addresses needed for the optimal strategy over arbitrary time, as the cycle effectively repeats itself with no differences.

To make this point clearer, let us consider what happens for an optimal strategy in one cycle. The address whose cycle we are consider, takes up a whitelist registration, and a session at the same time. As the session expires, the attacker registers another address and takes up a session, whilst the first address is in cool-off. This continues on until the first address' cool-off is over and it takes up another session, or the first address' whitelist period is over and other addresses take up the remaining sessions until the end of the first address' refund holding period is over. When the refund holding period is over, the first address returns to the first step, taking up a whitelist and session, effectively repeating the cycle with no differences. Therefore, we can easily infer the result of arbitrary time by analysing within one cycle.

In a $c + d$ period, for an optimal strategy, from 3.2.3 we know that there are $\lceil \frac{c+d}{b} \rceil$ sessions that has to be taken, and from 3.2.2.1 we know that each address can only take up $\lceil \frac{c}{a+b} \rceil$. Therefore, the minimum number of addresses is

$$\left\lceil \frac{\lceil \frac{c+d}{b} \rceil}{\lceil \frac{c}{a+b} \rceil} \right\rceil = \left\lceil \frac{\frac{c+d}{b}}{\lceil \frac{c}{a+b} \rceil} \right\rceil$$

□

Theorem 3.2.5. *It is impossible for a malicious actor who has a balance of X, who is using optimal strategy, to perform a successful distributed denial of service attack if* $\left\lceil \frac{\frac{c+d}{b}}{\lceil \frac{c}{a+b} \rceil} \right\rceil > \lfloor \frac{X}{Y} \rfloor$, *where a is the session period, b is the cool-off period, c is the whitelist period, d is the refund holding period and Y is the registration fee.*

Proof. By 3.2.4, we know that the minimum number of addresses necessary to carry out the optimal strategy for a successful distributed denial of service attack is $\left\lceil \frac{\frac{c+d}{b}}{\lceil \frac{c}{a+b} \rceil} \right\rceil$. We also know from 3.2.1 that the maximum number of addresses an actor can have is $\frac{X}{Y}$. If the minimum amount of addresses needed for a successful attack is higher than the maximum number an actor can have, then there will necessarily be gaps as the attack is performed. Therefore, the distributed denial of service attack would not be considered successful. □

As we can see from the theorem, that given the right adjustment of the config that gives the relation needed in 3.2.5, we can actually prevent a successful distributed denial of service attack, hence proving the safety property.

3.2.2 Gas and Economic Costs

In this section I will perform an analysis regarding the gas and economic costs of using the locking system in my implementation. It will not be the minimum possible gas used as the implementation was written with clarity as a priority, but it will be a good enough approximation.

Operation	Contract without Locking	Contract with Locking	Diff
Contract Creation	1705943	167515	1538428
Setting State Variable (1st Single)	152433	41709	110724
Setting State Variable (2nd Single)	28544	26709	1835
Setting State Variable (1st Dynamic)	167946	42030	125916
Setting State Variable (2nd Dynamic)	29057	27030	2027
1st Whitelist Register	43541	N/A	N/A
2nd Whitelist Register	28541	N/A	N/A
Unlocking 1 lock	49936	N/A	N/A
Unlocking 3 locks	66315	N/A	N/A

It is important to note that there is a need to evaluate and compare the first and later calls for the operations as all of the operations involve storing values to the smart contract, and there is a difference in gas cost between initially setting a value and modifying an already set value.

3.2.2.1 Explaining the differences

3.2.2.1.1 Contract Creation There are significantly more routines and state variables needed to use the Locking library via the inheritance of the **Lockable** contract. Deploying the **Lockable** contract by itself would have cost 957213 gas.

3.2.2.1.2 Setting State Variables Let us focus on the 1st operation at setting state variable. There is an approximately 120000 difference in gas

cost between a contract with locking and without locking. This can be explained by the storage of values in LockRecord and UserRecord, which takes up to an additional 6 **SSTORE** opcodes costing 20000 each. We can see it more clearly when we evaluate the 2nd operation, and the difference drops to a negligible amount.

$$1 + \frac{a}{b} + \frac{d}{c} + \frac{a}{b}\frac{d}{c} < \frac{x}{y}$$

Chapter 4

Discussion

4.1 Usability Friction

It is quite evident that there is a non-negligible amount of usability friction that arises from the use of the locking system and the various recovery and alleviation mechanisms. This cost is unavoidable as synchronization primitives add overhead in all occurences, smart contracts or otherwise. It is a trade-off for getting more correctness and safety into the smart contract, and should be considered with those properties in mind. The backward immutability of blockchain and smart contracts necessitates that these properties are prioritized.

4.2 Resource Costs

From 3.2.2 we can see that there is a non-trivial amount of additional resource cost when utilizing the locking system. Note that this is a proof-of-concept implementation that is focused on displaying the safety and liveness properties of the locking system. These resource costs can be reduced significantly through optimization, be it through a pre-processor or an implementation of the locking system as a standalone smart contract service.

There can also be protocol or virtual machine level optimization through introducing new routines specific to synchronization primitives and extending address properties. Given how gas is currently metered with relation to actual computational resource usage in nodes, this would significantly reduce the resources needed on the blockchain layer.

4.3 Liveness and Distributed Denial of Service

The recovery and alleviation mechanisms have been shown to incentivize and thus improve on liveness in the system, and a proof was shown in 3.2.1.3 that given the right config, a distributed denial of service attack can be defeated. It may appear that the config appropriate to defeat the distributed denial of service attack can drastically increase the resource and friction cost in using the system. However, it should be noted that the config can be modified whilst the smart contract is already deployed. This means that the distributed denial of service protection config can be *turned on* when necessary, and allowing a less punitive config when there is not an attack.

It is important to also note that the effectiveness of the mechanisms grow as the duration of the attack increases, as the resource balance of the malicious attacker would gradually reduce over time due to transaction and deposit costs. Thus, making the protection a "war of attrition" which only makes the smart contract grow in value(due to the deposits going to the contract) and liveness over time.

Chapter 5

Related Work

5.1 Adding concurrency to smart contracts

This paper [3], published in 2017 by Dickerson et. al proposes a way to detect concurrency conflicts through techniques from software transactional memory applied at validation time, with the intention of speeding up mining and validation through concurrency. Our research differs from this in where we focus our evaluation on concurrency – the actual execution of smart contracts for them, and the correct execution of intentional *business logic* on smart contracts for ours. That being said, there are methods being utilized by Dickerson et. al that can be integrated into future research.

Chapter 6

Conclusion

I have thus far shown in this paper that introducing synchronization primitives such as mutexes into smart contracts creates non-trivial liveness issues that require additional extension mechanisms in order to alleviate and circumvent. The mechanisms themselves are interesting cryptoeconomics topics on their own and may have further applications in areas outside of synchronization.

The complete proposed locking system is a proof-of-concept that is currently unfeasible, cost-wise, to use as at a production level. Further improvements can be done in optimisation through implementing it as a smart-contract-as-a-service or introducing it on a protocol or virtual machine level, before it is ready to be used. That being said, the value of this paper is in the evaluation of synchronization primitives when introduced to smart contracts and the potential safety and correctness benefits it brings to the table. With these properties, it is not too difficult to see that the exploits and bugs, that occured in the short history of smart contracts and will occur in the future, could and can be protected against if these primitives were to be used.

Bibliography

[1] *Block King Contract*. https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1.
Last Accessed: 2019-03-18.

[2] *DAO*. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee.
Last Accessed: 2019-03-18.

[3] Thomas Dickerson et al.
"Adding Concurrency to Smart Contracts". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. PODC '17. Washington, DC, USA: ACM, 2017, pp. 303–312.
ISBN: 978-1-4503-4992-5. DOI: 10.1145/3087801.3087835.
URL: http://doi.acm.org/10.1145/3087801.3087835.

[4] *Ethereum Gas Model*.
https://github.com/ethereum/wiki/wiki/Design-Rationale.
Last Accessed: 2019-03-18.

[5] *Java Synchronized*. https://www.baeldung.com/java-synchronized.
Last Accessed: 2019-03-18.

[6] *OpenZeppelin Re-entrancy*.
https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/utils/ReentrancyGuard.sol.
Last Accessed: 2019-03-18.