BTC Architectural Enhancement

Makayla McMullin
19mam51@queensu.ca

Aniket Mukherjee
aniketm7274@gmail.com

Daniel Dickson
d.dickson@queensu.ca

Maia Domingues
18mkd6@queensu.ca

Lucas Patoine
19lwp@queensu.ca

Devon Gough
d.gough@queensu.ca

2023-04-12

Contents

1 Abstract

In this paper we evaluate how the architecture of the Bitcoin Core software would change if a Vault enhancement were added. We detail an overview of the enhancements, as well as showcase two possible ways of implementing the changes to the software. We explain who the shareholders are, and what they are looking for with this enhancement. Finally, we go over the potential risks of implementing a system like this and how we would set up tests to mitigate as many of those risks as possible.

2 Introduction

The Bitcoin Core software already comes equipped with a handful of features designed to keep users as well as their funds as safe as possible from potential attacks. However, despite the protections that have already been implemented, users can still fall victim to sophisticated attacks designed to steal their bitcoin. A vault enhancement can protect users from attacks. One of the main types of attacks that the Vault is designed to protect against is social engineering attacks, where an unauthorised person gets access to a wallet's private key by convincing the user to share it with them. By implementing this new feature to Bitcoin Core, we can help prevent this. The vault is designed as a fail-safe wallet that notifies users when funds are moved out of an account, giving them a designated number of blocks to revoke the transaction and redirect the funds to the specified recovery wallet, reducing the potency of potential attacks.

3 Enhancement Overview

In its current implementation, Bitcoin Core offers a good variety of security features designed to keep the network and the user alike safe in their operations. The network safety is maintained with the proper validation of blocks and transactions, in addition to the decentralized nature of the blockchain ensuring everyone follows the rules. The users are kept safe through the encrypted `wallet.dat` file, the public/private key system in combination with hierarchical deterministic wallets. These safeguards alone are not enough to prevent users' funds from being accessed through various means, including inadvertently sharing a private key.

The Vault enhancement adds an additional layer of security to Bitcoin Core in the form of an alert that gets broadcast network-wide when funds are being moved out of a user's account. This is implemented via a new operation code (op-code), detailed in the next section. The Vault is created with a few parameters, namely the unvault key and the recovery address. The unvault key is used to authorize the extraction of bitcoin from the wallet, and the recovery address is where the bitcoin will go if a transaction is marked as fraudulent. The system works similarly to a smart contract, where the Vault controls where the bitcoin it holds can be sent.

If a user opts to use a vault, they must broadcast 2 separate transactions in two separate blocks before being able to spend their BTC, which in turn will allow users to give the all clear to any transaction they make, or sweep their bitcoin into a different wallet, all before the transaction goes through completely. This would be able to stop the worst outcome of a user's key being compromised, and save the user a lot of stress in the event of such a breach in security. To implement this enhancement, a backwards compatible change to the blockchain could be implemented which, if adopted, would be represented in the operational code of Bitcoin Core.

3.1 Possible Methods of Implementation

The current Bitcoin Core system is a Publication-Subscription architectural style. The below graph details the components of the systems and the dependencies of each component. There are two relevant methods of implementation for a Vault enhancement on the Bitcoin Core system: modifying pre-existing modules to perform the enhancement or implementing the enhancement using new modules. We will discuss both options in detail then analyse them using the SAAM method to determine which implementation is more satisfactory to the stakeholders.

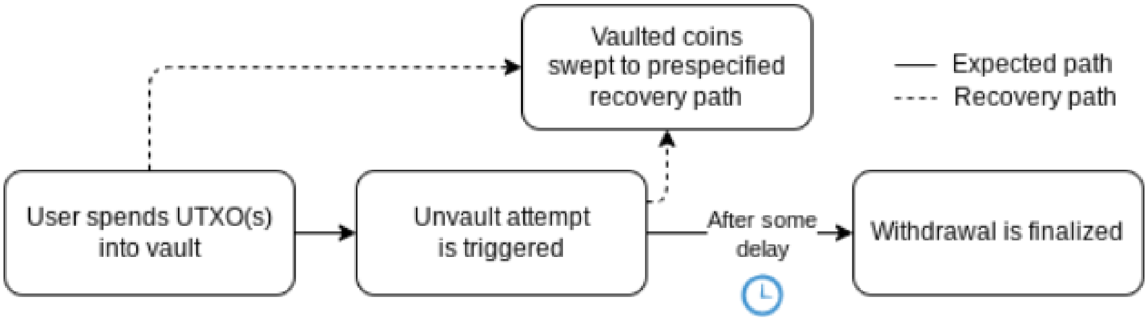

Vaulted coins swept to prespecified recovery path
Expected path
Recovery path
User spends UTXO(s) into vault
Unvault attempt is triggered
After some delay
Withdrawal is finalized

Figure 1: Vault Diagram Description [2]

3.2 Alternative 1: *Implementation Using CTV Among Existing Modules*

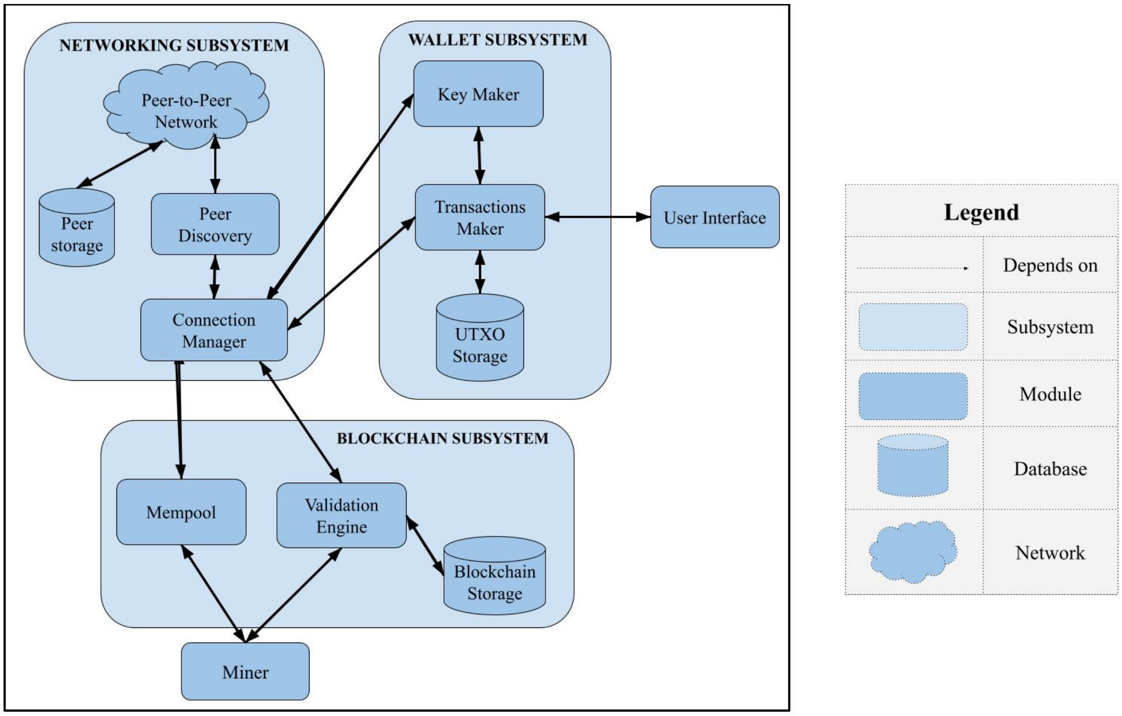
One way that our proposed vault enhancement could be implemented into Bitcoin Core would be to implement the vault functionality over the pre-existing modules within the software architecture. This approach would require fundamentally changing the way that each of the current modules within a relevant subsystem function to allow for funds to be sent to the vault, either when requested by the user or when prompted due to a possible attack being performed on the user's wallet. This proposed vault enhancement method will mainly focus on implementing what is known as CheckTemplateVerify (CTV) to create a vault-like structure on which the vault functionality can be built on top of. Through this implementation, the Publish-Subscribe architectural style would be preserved.

To implement a vault into Bitcoin core over pre-existing modules using CheckTemplateVerify, The first step would be to add a new operation code, `OP_CHECKTEMPLATEVERIFY` [1] . This operation code will allow for the verification and enforcement of various security features and recovery fail-safes to make sure that attacks on bitcoin funds within a user's wallet are more difficult to carry out. Some examples of specific security features that the CheckTemplateVerify opcode could employ are requiring multiple signatures to authorize withdrawal, having a time lock put in place so that funds can only be spent after a certain time, and creating a template of future transaction hash serials that must be matched by future transactions for them to be verifiable.

The next step for implementation would be to modify the pre-existing modules within the wallet and blockchain subsystems, when appropriate, to add support for managing transactions within a vault and to prepare for CTV opcode implementation. Because these modules were not made with the intention of supporting vault functionality, support will need to be added to ensure proper co-operability between new and existing code, and adherence to the suggested upgrades.

Once support is added, developers can now apply what has been defined under the CTV operation code to the affected modules, therefore implementing vault functionality. In this step, the code of each relevant module will be edited to reflect the conditions defined within the CTV opcode. For example, when a user wishes to spend funds stored in their vault, the safety features should engage and only allow for spending to occur under the specific conditions defined within that operation code. These security enforcements will be the underlying structure of the vault implementation without creating a specific vault module within the software architecture.

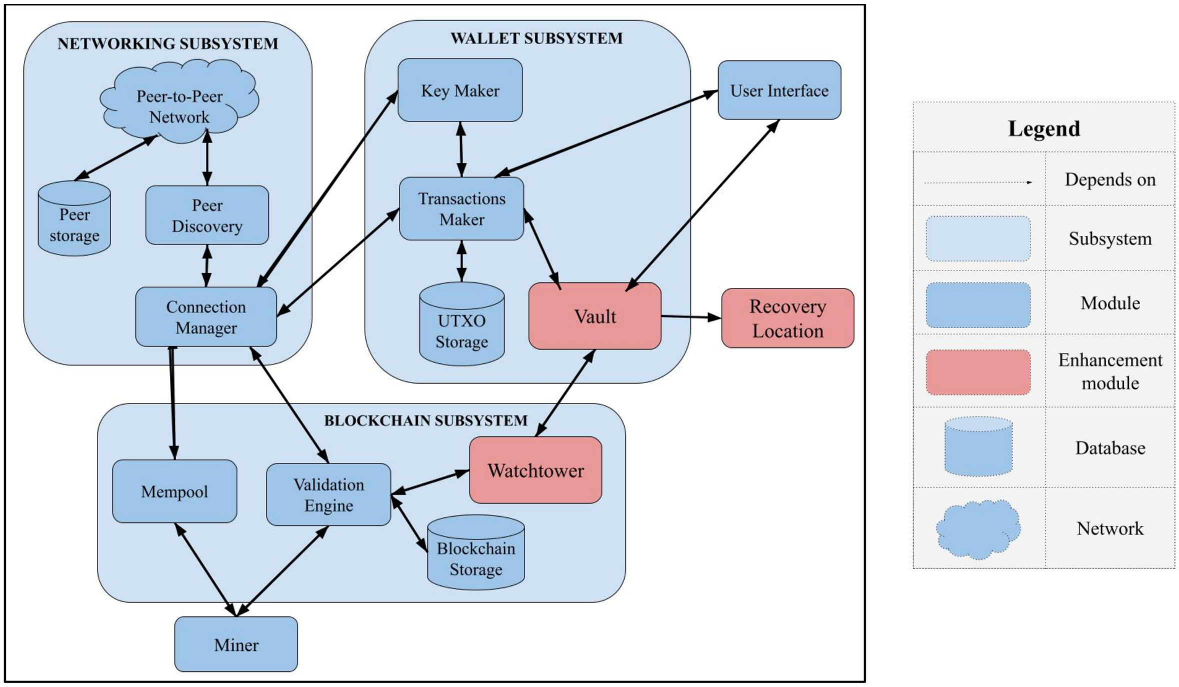
A possible drawback of this implementation is that there is less flexibility than if one singular module is created for vault functionality, as each module in an affected subsystem may need to be edited accordingly and, in the case of an error occurring, there are more points of failure. Developers may have a difficult time implementing changes and updates when code is scattered across modules rather than in one central place.



3.3 Alternative 2: *Implementation Using New Modules*

Another way to implement a vault enhancement is to add additional modules to the system design. Adding new modules allows the other modules in the system to stay unchanged, which helps with testing because smaller parts of code can be tested and modified without needing to change other components and functions. This implementation of vaults also helps the overall modularity of the system. Furthermore, there is no change to the overall structure of the architecture. The system remains a Publish-Subscribe architectural style. Leaving the architectural style unchanged avoids any compatibility problems and new bugs that could develop with a change in architectural style.

As shown in the dependency diagram below, this implementation requires three new modules: a Vault module, a Recovery Location and a Watchtower module. The Vault module receives UTXOs from the transaction Maker and stores them. The Vault sends a list of the UTXOs in the Vault and the Watchtower module. The Watchtower module monitors the blocks passing through the Validation Engine for any UTXOs that are in the Vault. If it sees a UTXO in the blockchain that is also in the Vault it sends an un-vaulting attempt notification to the Vault module. Whenever a vault transaction is detected, the Vault sends a message to the user (through the UI) asking for approval. The user's answer is sent back to the Vault from the UI. If the user does not approve the transaction, all UTXOs are sent to the Recovery Location. If the transaction is approved, the Vault will wait a specified amount of time and then send a message approving the transaction to the Transaction Maker.



3.4 SAAM Analysis

The following section presents a SAAM architectural analysis between the two different approaches being considered for the implementing a vault feature to our proposed architecture of Bitcoin Core. It compares major stakeholders, important NFRs, as well as the impact implementing such an enhancement in each way would have. This information is then used to determine which approach would be preferred for the implementation of this feature.

The major stakeholders we identified are the Users and Programmers. The main NFRs that we believe the users will focus on are security, usability, performance, and compatibility/portability, while the programmers will likely care the most about maintainability, testability, and extensibility. The NFRs identified did not change with the two different implementations.

The users' most important non-functional requirements for the enhancement are security, usability, performance, compatibility and portability. Security is important to the users because the system holds all of their currency. If there is a security breach users could lose a lot of money. Usability is important to the users because they want to easily navigate the system, conveniently manage their cryptocurrency without errors and for the system to run efficiently on their host system. Performance is important to users because they will want to make transactions quickly without unnecessary lag. Compatibility and portability are important to users because they will require the system to run smoothly across multiple environments and work well with pre-existing systems on their host computer.

The programmers' most important non-functional requirements are maintainability, testability and extensibility. Since Bitcoin Core is an open-source project, the system must be easily maintainable so that it can be done collaboratively without needing specific programmers for tasks. Similarly, the enhancement must be easy to test so that it can be done by any contributing programmer. Extensibility is also important to programmers since Bitcoin is an international and extensive technology that should have room to grow and adapt alongside global technological advancements.

3.5 Impact on NFRs and Stakeholders

3.5.1 Users

	Alternative 1	**Alternative 2**
NFR		
Security	Because the CheckTemplateVerify opcode script is visible on the blockchain for each transaction, if proper security measures are not taken to obfuscate sensitive and revealing information on how the vault is implemented, hackers may be able to intercept this data and compromise the overall system, dampening the intention of enhancing blockchain security for users.	The enhancement's security is an improvement over the original system but still has vulnerabilities. Security is reliant on strength of recovery location so if the recovery location is compromised the whole system is compromised.
Usability	With a CheckTemplateVerify-based vault, users would be notified when a potentially fraudulent transaction is occurring according to the security checks in place by the CTV opcode. This may steepen the learning curve for new users trying to learn how to properly use and set up a Bitcoin vault within Bitcoin Core.	With the increased amount of messages being transferred between modules, the efficiency of the system would be lowered. This would make the system use more power on the user's host computer.
Performance	Overall performance of the user's wallets may be impacted due to the wallet subsystem needing to process transactions based on the security checks determined by CheckTemplateVerify, in addition to regular transactions.	The throughput of the system could be limited by the increased number of dependencies through the addition of new modules. The more dependencies a module has the more time it spends waiting for signals or work from those dependencies before it can perform its own work.
Compatibility	Because this vault implementation option involves changing the code of existing modules, the overall codebase will be substantially changed and require users to update their software to reflect the new changes. Compatibility issues may arise between different Bitcoin Core versions as users with different versions of the software (i.e. processing a transaction between someone with an older version of the software and someone with a version reflection the vault change) try to transact with one another, which may prove especially complex due to the peer-to-peer nature of the Bitcoin Core network.	Since there is no change to the existing components in the system the portability and compatibility of the system is unchanged. Those who want the enhancement, can easily download the enhancement modules separately and have them interact well with the system they already have installed.
Overall Impact	While users in most cases may not notice a substantial difference in how Bitcoin Core would operate on a daily basis and would appreciate the enhanced security measures a CTV-based vault would provide, a steeper learning curve paired with potential major compatibility concerns make this implementation difficult to justify for users	Usability and performance may be slightly diminished because of the computational power required by the implementation's modularity. However, the portability and compatibility of Bitcoin Core will be unaffected by this implementation. So while this implementation does add computational overhead, all users will be able to easily access it.

3.5.2 Programmers

	Alternative 1	**Alternative 2**
NFR		
Maintainability	Spreading the features throughout the existing modules will decrease maintainability since the functions will be added onto existing classes and files, making them larger and responsible for more functionality. However, introducing new code over multiple modules will create many points of failure if an issue occurs within the vault functionality, making it more difficult to find and fix bugs.	The modularity of this implementation helps with maintainability as functions are separated into different sections, making the code readable and less robust. Understandable and less complex code is easier to maintain .
Testability	Testing code will likely be harder to implement, as we would need to add new tests to cover the newly implemented files in addition to changing tests in many different modules, files, and classes to achieve the same test coverage.	Since enhancement features are separate from other modules they can be tested separately. The programmers will know that the older modules in the system will not be affected so they do not have to waste time retesting them .
Performance	Overall performance of the user's wallets may be impacted due to the wallet subsystem needing to process transactions based on the security checks determined by CheckTemplateVerify, in addition to regular transactions.	The throughput of the system could be limited by the increased number of dependencies through the addition of new modules. The more dependencies a module has the more time it spends waiting for signals or work from those dependencies before it can perform its own work.
Extensibility	If we wanted to extend the functionality of the enhancement it would require tracking down what changes are already implemented in each file that was initially changed when the feature was launched, then ensuring the new functionality does not conflict with the other duties of the module/class/file being changed.	The modularity of this implementation allows for modules to be evolved and upgraded separately from the other modules without affecting them.
Overall Impact	This is definitely not the right implementation for programmers, having code for this functionality spread all over the place reduces security and increases the effort required to both implement features and do the required maintenance.	Maintainability, testability and extensibility all benefit with this implementation of the Vault enhancement. Making this alternative an attractive option for the programmers.

3.6 Preferred Implementation of Architectural Enhancements

After evaluating the pros and cons as well as conducting a SAAM analysis of the two different approaches, we came to the conclusion that implementation 2 is the preferred method. This decision was made primarily based on the ease of implementation on the developer side. The user experience with both implementations is essentially identical, with a possible slight boost to performance if implementation 2 is followed. For the developers and engineers working on the project, the upsides of having all code responsible for a given feature all in one place cannot be understated. It makes the system as a whole far less confusing, decreasing the time taken for a new developer to get up to speed. In addition, the developers do not need to worry that they might accidentally interfere with the already completed modules, since instead of bouncing around between many different subsystems to implement the feature, all work will be done in the same spot. Finally, if the feature were to be improved upon in the future, having the feature implemented as its own module greatly reduces the complexity required to implement the updates.

4 Effect on Quality Attributes

As a new feature is being implemented, this will ultimately result in some effects on the maintainability, evolvability, testability, and performance of the Bitcoin Core system. Maintainability, a measure of how easily the system can be understood or developed, will decrease. This is as a result of additional components being added into the system, resulting in an increase of processes, dependencies, code files, etc. for developers to understand and have to maintain as the project continues. The more complex a system gets, the more difficult it becomes to maintain. Testability refers to how effective testing a system is with regard to the ability to easily test it and receive effective and useful results from those tests. The amount of testing for the overall system will definitely increase, due to the additional components that now require regular testing to ensure their continued functionality. The tests themselves should be fairly straightforward. Our new feature improves cybersecurity, a field for which there are many testing procedures already in place in most systems, thus a major change in overall system testing will be seeing how the system handles cyber attacks, and that the vault is correctly implemented and used. More information will be discussed in the following section of our report.

Performance is a measure of how effectively a system runs, especially regarding its responsiveness, stability, and ability to function as the developers intended. The overall performance of the software may be slowed due to having to run our extra feature frequently, as checking for potentially fraudulent transactions should be done for every transaction. However, the system's ability to function properly is increased, as the chance for error in the form of allowing unauthorized transactions is mitigated. Some of this is also mentioned in more detail in the Potential Risk section of our report. Evolvability evaluates how well the system can be adapted to fulfil new requirements in the future, that is, how it can evolve to meet future user needs. By keeping all of our new additions to their own separate modules, it will be easier for the developers to keep track of where the code for this feature is stored in the system. Thus, adding on future capabilities or enhancements remains straightforward, as the developers can still modify only necessary components instead of having to search through several components to get to the necessary source code.

5 Testing Plans

The testing process for the two different implementations of the Vault are not going to be the same. The first alternative (where the functionality of the improvement is spread out across existing modules) will require editing the existing tests of the modules being changed. This will mostly happen with unit testing, where each function will have its boundaries in place before the code is finished (ideally, before the code is even started). Since this implementation requires editing already existing code, this will be more confusing for the programmer and likely consist of additional work keeping track of what files/classes interact with one another. The functionality of these modules will need to be backwards-compatible, meaning most of the tests that passed before the implementation of the enhancement should also pass after. This happens through integration testing, where modules as a whole are tested, to ensure that they work well with one another, and produce the expected outcome. The interactions between modules is also tested, e.g. there should be a test that checks that the UI calls the proper method for creating a transaction and ensures the proper UI response.

If the improvement is implemented using alternative 2 instead, the testing strategy changes. The feature is implemented as its own module, leaving all the existing code and tests alone. This allows more concise and simple files to be written all in one place, decreasing the chance that some functionality slips through untested due to programmer error. The time to implement the tests will also decrease with this implementation, as the existing modules only need to be updated with more integration tests, rather than adding more unit tests.

The testing method likely to be used is white box testing, where the engineer chooses an array of inputs and the corresponding outputs that it should provide, and checks to verify the unit/module/class is working as intended.

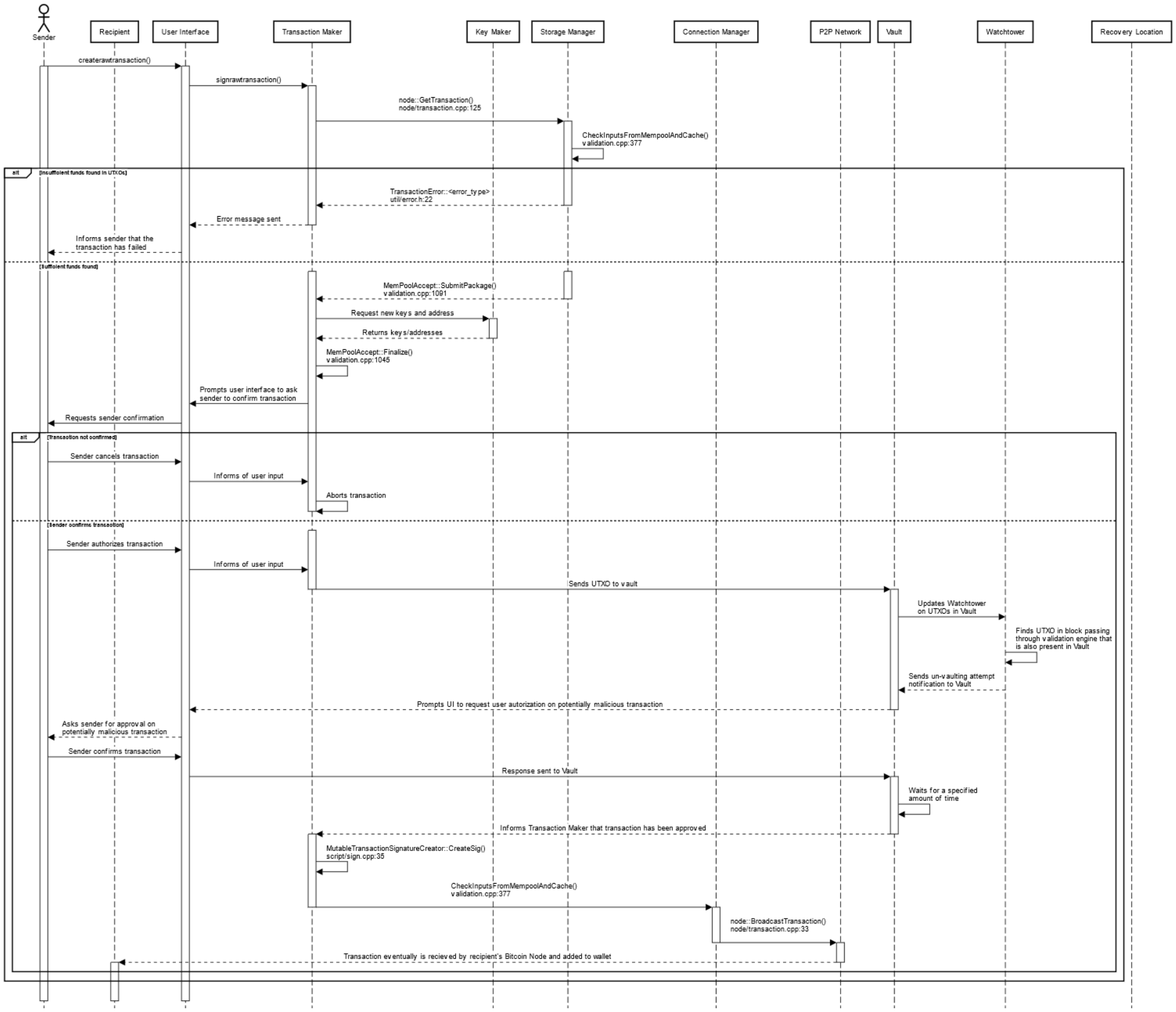
6 Potential Risk

Due to the extra security measures that a vault would require(the manual confirmation of any and all transactions from one wallet to another), it would slow the speed of many transactions that Bitcoin Core handles, as it would no longer automatically go through. Speed is also reduced further in the event of an actual breach where funds are diverted to an alternate wallet, as it's an additional process that occurs that Bitcoin Core is going to need to deal with. Furthermore, there's also the scenario where the alternative wallet is also compromised, which means that even if the user chooses to sweep their coins into another wallet to protect them, it might only be a temporary security solution to their initial wallet being compromised, leaving their coins vulnerable to theft yet again.

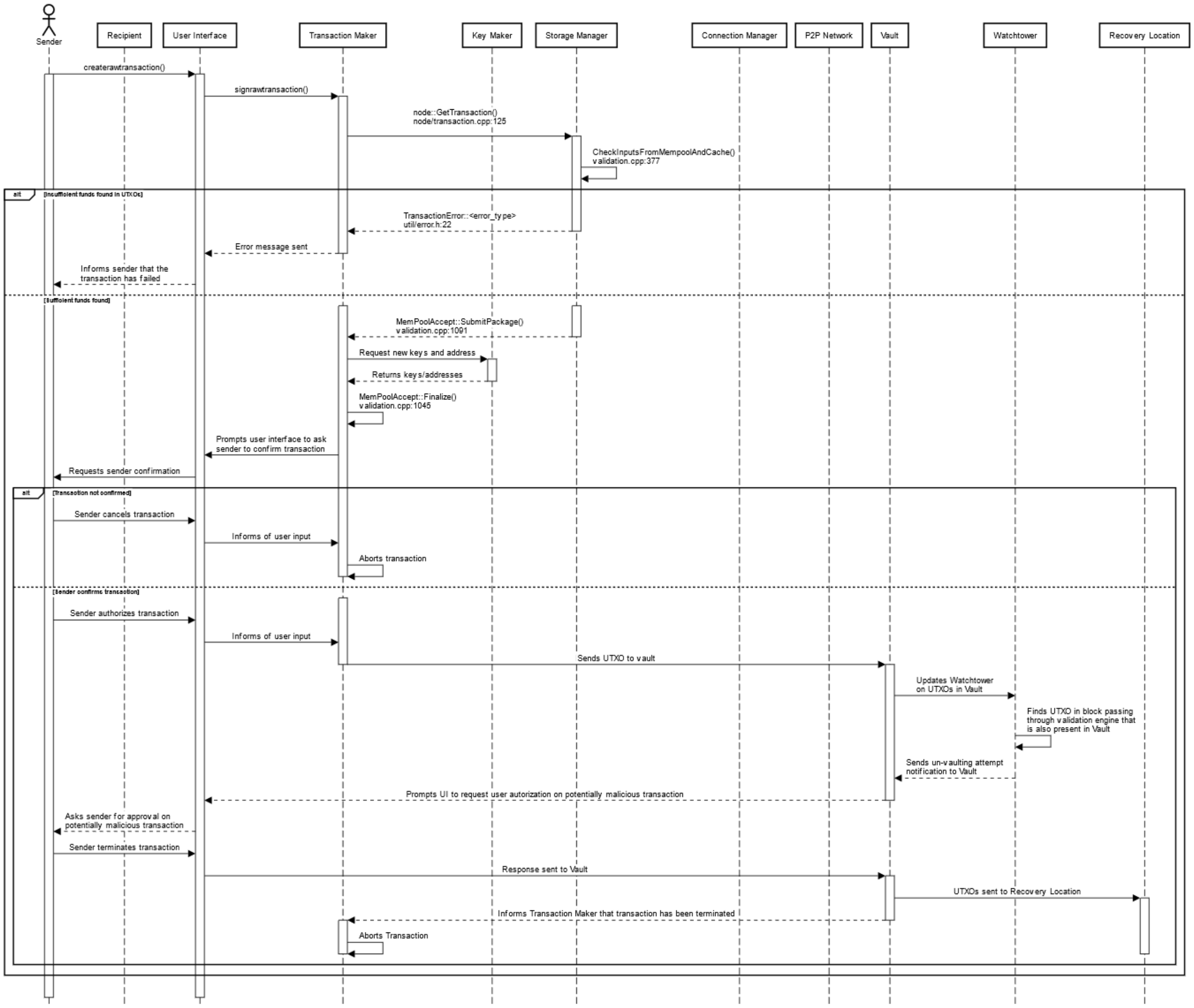
7 Use Cases

Based on the preferred method of implementing a vault feature into Bitcoin Core, what follows are two different use cases that directly relate to this new feature, along with sequence diagrams that demonstrate the interactions between different modules, including the new ones introduced in implementing the vault.

Case 1: A fraudulent transaction is made, the user is informed, and terminates it.


Sender
Recipient
User Interface
Transaction Maker
Key Maker
Storage Manager
Connection Manager
P2P Network
Vault
Watchtower
Recovery Location
createrawtransaction()
signrawtransaction()
node::GetTransaction() node/transaction.cpp:125
CheckInputsFromMempoolAndCache() validation.cpp:377
alt
[Insufficient funds found in UTXOs]
TransactionError::<error_type> util/error.h:22
Error message sent
Informs sender that the transaction has failed
[Sufficient funds found]
MemPoolAccept::SubmitPackage() validation.cpp:1091
Request new keys and address
Returns keys/addresses
MemPoolAccept::Finalize() validation.cpp:1045
Prompts user interface to ask sender to confirm transaction
Requests sender confirmation
alt
[Transaction not confirmed]
Sender cancels transaction
Informs of user input
Aborts transaction
[Sender confirms transaction]
Sender authorizes transaction
Informs of user input
Sends UTXO to vault
Updates Watchtower on UTXOs in Vault
Finds UTXO in block passing through validation engine that is also present in Vault
Sends un-vaulting attempt notification to Vault
Prompts UI to request user autorization on potentially malicious transaction
Asks sender for approval on potentially malicious transaction
Sender confirms transaction
Response sent to Vault
Waits for a specified amount of time
Informs Transaction Maker that transaction has been approved
MutableTransactionSignatureCreator::CreateSig() script/sign.cpp:35
CheckInputsFromMempoolAndCache() validation.cpp:377
node::BroadcastTransaction() node/transaction.cpp:33
Transaction eventually is recieved by recipient's Bitcoin Node and added to wallet

Case 2: The user makes a transaction, is alerted, and approves of it.


Sender
Recipient
User Interface
Transaction Maker
Key Maker
Storage Manager
Connection Manager
P2P Network
Vault
Watchtower
Recovery Location
createrawtransaction()
signrawtransaction()
node::GetTransaction()
node/transaction.cpp:125
CheckInputsFromMempoolAndCache()
validation.cpp:377
alt
[Insufficient funds found in UTXOs]
TransactionError::<error_type>
util/error.h:22
Error message sent
Informs sender that the
transaction has failed
[Sufficient funds found]
MemPoolAccept::SubmitPackage()
validation.cpp:1091
Request new keys and address
Returns keys/addresses
MemPoolAccept::Finalize()
validation.cpp:1045
Prompts user interface to ask
sender to confirm transaction
Requests sender confirmation
alt
[Transaction not confirmed]
Sender cancels transaction
Informs of user input
Aborts transaction
[Sender confirms transaction]
Sender authorizes transaction
Informs of user input
Sends UTXO to vault
Updates Watchtower
on UTXOs in Vault
Finds UTXO in block passing
through validation engine that
is also present in Vault
Sends un-vaulting attempt
notification to Vault
Prompts UI to request user autorization on potentially malicious transaction
Asks sender for approval on
potentially malicious transaction
Sender terminates transaction
Response sent to Vault
UTXOs sent to Recovery Location
Informs Transaction Maker that transaction has been terminated
Aborts Transaction

8 Data Dictionary

Publish-and-Subscribe (Architectural Style): *(also, Implicit Invocation)* A software architectural style suitable for projects that involve a loosely-coupled collection of components, which carry out some operation and possibly enable other operations. A broadcasting system may invoke necessary procedures as they are needed.

9 Naming Conventions

UXTO (Unspent Transaction Output): Technical term for an amount of bitcoin that in a transaction.

10 Limitations and Lessons Learned

Overall, our group learned a lot about the importance of SAAM evaluation in a software system, and how to measure the system's overall quality. We also found that we were rather limited in our ability to research our proposed feature. This was due to two main factors: firstly, as per our previous report, finding completely up-to-date information about the architecture and code analysis of Bitcoin Core was very difficult, and secondly, that this is a completely new feature we are suggesting to implement while only having examined Bitcoin Core's architecture for about two months, compared to developers who have been maintaining and contributing to the project for years. Additionally, many of these topics were not covered as thoroughly in class, so we had to do a lot of research on what was being asked, as well as having to figure out the implementation of our new feature.

While examining the possibility of our new feature, we learned a lot about how relatively small changes can have a large impact in a system. We also got to explore several distinct ways of implementing the same change into the system, and how these ways can drastically change the software's architecture and structure or barely affect it at all, despite (ideally) resulting in the exact same overall functionality. We also got to examine how the quality attributes of a system change when new features are implemented. Though it objectively makes sense, it was interesting to see how many of these attributes were impaired by the addition of a new feature to the system, as added complexity makes the system overall harder to maintain and test, and adds more stress to the system's performance. It was also very valuable to get to see how much care needs to go into implementing new features, as this process and our newfound understanding will surely be a useful skill to carry over to future projects.

11 Conclusions

Throughout this report we proposed and explored different ways in which a vault feature could be implemented in Bitcoin Core, and how the architecture we developed in the two prior reports would change as a result. This enhancement would help add an extra layer of security when it comes to transferring funds, since as it stands, if someone gained access to a wallet's private key, there would be nothing stopping them from making transfers without the owner's consent. Two distinct approaches were considered for the implementation of this enhancement. One approach involved the modification of pre-existing modules as well as the use of CheckTemplateVerify to create a vault-like structure. This approach would come with sizable changes to the functionality of many modules, however would be less impactful towards the overall architecture. The other approach was more direct, involving the addition of the modules: Vault, Watchtower, and Recovery location. Each module represents a key aspect of the proposed vault enhancement and introduces new interactions between modules and subsystems. In the end we decided that the latter approach would be favourable for the implementation of the proposed enhancement, as not only did it allow for easier maintainability, testability, and extensibility on account of its modularity, but the other approach reduced security, and made maintenance much harder. Overall, we believe that through the methods discussed, we could feasibly introduce a vault feature to our proposed architecture of Bitcoin Core that gives users an additional layer of security, while remaining non-intrusive to the process of making transactions.

12 References

1. "What is OP_VAULT and how will it benefit bitcoin users?: River Learn - Bitcoin Technology," River Financial, 2023. [Online]. Available: https://river.com/learn/what-is-op-vault-in-bitcoin/. [Accessed: 09-Apr-2023].

2. J. O'Beirne, "Vaults and Covenants - jameso.be," Vaults and Covenants, 23-Jan-2023. [Online]. Available: https://jameso.be/vaults.pdf. [Accessed: 12-Apr-2023].