SMART CONTRACT AUDIT REPORT

for

Metatime

Prepared By: Yiqun Chen

PeckShield

December 20, 2021

Document Properties

Client	Metatime
Title	Smart Contract Audit Report
Target	Metatime
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 20, 2021	Jing Wang	Final Release
1.0-rc	December 15, 2021	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Metatime protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Metatime

Metatime is a SocialFi project built on Binance Smart Chain (BSC). It is a metaverse project based on Time and Matter with the purpose of encouraging people to cherish time and create value for their time. Compared to other metaverse projects, Metatime has no tourist mode. Also, a complete digital identity is required to enter Metatime, including a wallet address, as well as NFT and the metaverse time exchanged by Time token.

The basic information of the Metatime protocol is as follows:

Table 1.1: Basic Information of The Metatime Protocol

Item	Description
Name	Metatime
Website	https://metatime.social/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 20, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/MetatimeSocial/metatime.git (6fd2d5b)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/MetatimeSocial/metatime.git (60a211b)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Metatime` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	
Low	3	
Informational	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Metatime Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Handling of isActive in UserProfile::_withdraw()	Business Logic	Fixed
PVE-002	Medium	Proper Handling of reward() in RewardTheAuthor	Business Logic	Fixed
PVE-003	Low	Possible Overflow Prevention With SafeMath	Coding Practices	Confirmed
PVE-004	Low	Duplicate Pool Detection and Prevention In MutiRewardPool	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Low	Proper Handling of Unsupported Token in RewardTheAuthor::claim()	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper Handling of isActive in UserProfile::_withdraw()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: UserProfile
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the Metatime protocol, the UserProfile contract provides an interface for users to create profiles by depositing the supported NFT into the contract. The profiles will be deleted after the deposited NFT is withdrawn. To elaborate, we show below the _withdraw() function in the UserProfile contract.

```
    function _withdraw() internal returns(bool) {

        User storage u = Users[msgSender()];
        require(u.isActive, "not active");
        require(u.user_id != address(0), "has not deposited.");
        require(u.user_id == msgSender(), "not nft owner");

        uint256 tokenID = u.token_id;
        IERC721 nftToken = IERC721(u.NFT_address);
        nftToken.safeTransferFrom(address(this),msgSender(), tokenID);


        u.user_id = address(0);
        u.NFT_address = address(0);
        u.token_id =0;

        delete nicknames[u.nickname];

        emit WithdrawNFT(msgSender(), address(nftToken), tokenID, block.timestamp);

        return true;
```

```
    }
```

Listing 3.1: `UserProfile::_withdraw()`

We notice the deletion of a user profile is incomplete. The `u.isActive` is left as `true` after the calling of `_withdraw()`, thus a deleted user would still be able to call `updateNickname()` to change the `nickname`. A bad actor may reserve many `nicknames` by depositing and withdrawing a same `NFT` multiple times from different addresses.

Recommendation Improve the user profile deletion logic in `UserProfile::_withdraw()`.

Status The issue has been fixed by this commit: `75d27f8`.

3.2 Proper Handling of reward() in RewardTheAuthor

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `RewardTheAuthor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `RewardTheAuthor` contract provides a `reward()` routine for the user to send rewards to the author, and a `claim()` routine for the author to claim the rewards from the user. To elaborate, we show below the related routines.

```
    /**
     * @dev Reward the designated author
     * @param target the author
     * @param token the token to be rewarded
     * @param postType the post type
     * @param postId the post id
     * @param amount Amount to be rewarded
     */
    function reward(
        address target,
        IERC20 token,
        uint256 postType,
        uint64 postId,
        uint256 amount
    ) public payable {
        require(_supportTokens.contains(address(token)), "Unsupported token");

        if (msg.value > 0) {
            require(address(token) == address(_weth), "bad params");

```

```
            _weth.deposit{value: msg.value}();
            amount = msg.value;
        } else {
            uint256 oldBal = token.balanceOf(address(this));
            token.safeTransferFrom(msgSender(), address(this), amount);
            amount = token.balanceOf(address(this)).sub(oldBal);
        }

        require(amount > 0, "bad amount");

        uint256 pending = _userRewards[msgSender()][address(token)];
        _userRewards[msgSender()][address(token)] = pending.add(amount);

        _rewardId++;

        emit Reward(
            _rewardId,
            msgSender(),
            target,
            address(token),
            postType,
            postId,
            amount,
            block.timestamp
        );
    }
```

Listing 3.2: `RewardTheAuthor::reward()`

```
    function claim(address token) public {
        uint256 pending = _userRewards[msgSender()][token];
        if (pending == 0) return;

        _userRewards[msgSender()][token] = 0;
        _userClaimedRewards[msgSender()][address(token)] = _userClaimedRewards[msgSender
            ()][address(token)].add(pending);

        IERC20(token).safeTransfer(msgSender(), pending);

        emit Claim(msgSender(), token, pending);
    }
```

Listing 3.3: `RewardTheAuthor::claim()`

We notice the funds deposited into the contract via `reward()` is counted into `_userRewards[msgSender()][address(token)]` rather than `_userRewards[target][address(token)]`. This will only allow the users to withdraw the funds deposited by themselves, not by the rewarded author.

Recommendation Proper handling of reward() in the `RewardTheAuthor` contract.

Status The issue has been fixed by this commit: `be3d96b`.

3.3 Possible Overflow Prevention With SafeMath

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While analyzing the `CashierDesk` contract, we observe it can be improved by taking advantage of improved security from `SafeMath`. In the following, we use the `AddUsersBalance()` as the example.

```
168     function AddUsersBalance(
169         address token,
170         address[] memory users,
171         uint256[] memory values
172     ) public onlyCaller returns (bool) {
173         require(_support_token.contains(token) == true, "cant support token.");
174         require(users.length == values.length, "bad length");

176         for (uint256 i = 0; i < users.length; i++) {
177             _balanceOf[users[i]][token] += values[i];
178             emit AddToken(users[i], token, values[i], block.timestamp);
179         }

181         return true;
182     }
```

Listing 3.4: `CashierDesk::AddUsersBalance()`

From the above code, we notice that in the computation of `_balanceOf[users[i]][token] += values[i]` (line 177), the addition may result a number larger than `uint256`, thus would cause the updated `_balanceOf[users[i]][token]` overflow.

Also, we notice the requirement of `_index <= getMinterLength()- 1` (line 47) from `Meta::getMinter()` is not guarded against possible underflow. It is suggested to replace it with `_index < getMinterLength()`.

```
46     function getMinter(uint256 _index) public view returns (address) {
47         require(_index <= getMinterLength() - 1, "Token: index out of bounds");
48         return EnumerableSet.at(_minters, _index);
49     }
```

Listing 3.5: `Meta::getMinter()`

Recommendation Make use of `SafeMath` in the above calculations to better mitigate possible overflow or underflow.

Status The issue has been confirmed.

3.4 Duplicate Pool Detection and Prevention In MutiRewardPool

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MutiRewardPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Metatime` protocol has a `MutiRewardPool` contract that provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint *100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `addPool()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate `_stakingDuration` from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function addPool(
        uint256 _stakingDuration,
        uint256 _allocPoint
    ) public onlyOwner {
        massUpdatePools();

        // staking pool
        poolInfo.push(PoolInfo({
            lpToken: depositToken,
            totalDeposit: 0,
            duration: _stakingDuration,
```

```
            allocPoint: _allocPoint,
            lastRewardBlock: block.number > startBlock? block.number: startBlock,
            token0AccRewardsPerShare: 0,
            token1AccRewardsPerShare: 0,
            token0AccAdditionalRewardsPerShare: 0,
            token1AccAdditionalRewardsPerShare: 0,
            token0AccDonateAmount: 0,
            token1AccDonateAmount: 0
        }));

        totalAllocPoint = totalAllocPoint.add(_allocPoint);
    }
```

Listing 3.6: `MutiRewardPool::addPool()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(uint256 _stakingDuration) public {
        uint256 length = poolInfo.length;
        for (uint256 i = 0; i < length; ++i) {
            require(poolInfo[i].duration != _stakingDuration, "add: duration is already
                added to the pool");
        }
    }

    function addPool(
        uint256 _stakingDuration,
        uint256 _allocPoint
    ) public onlyOwner {
        massUpdatePools();
        checkPoolDuplicate(_stakingDuration);
        // staking pool
        poolInfo.push(PoolInfo({
            lpToken: depositToken,
            totalDeposit: 0,
            duration: _stakingDuration,
            allocPoint: _allocPoint,
            lastRewardBlock: block.number > startBlock? block.number: startBlock,
            token0AccRewardsPerShare: 0,
            token1AccRewardsPerShare: 0,
            token0AccAdditionalRewardsPerShare: 0,
            token1AccAdditionalRewardsPerShare: 0,
            token0AccDonateAmount: 0,
            token1AccDonateAmount: 0
        }));

        totalAllocPoint = totalAllocPoint.add(_allocPoint);
    }
```

Listing 3.7: Revised `MutiRewardPool::addPool()`

Status The issue has been fixed by this commit: 75d27f8.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Metatime protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., minting tokens , setting various parameters, etc.). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the mint() functions in the Meta token contract, which allows the Minter to add tokens into circulation and the recipient can be directly provided when the mint operation takes place.

```
    function mint(address _to, uint256 _amount) public onlyMinter {
        _mint(_to, _amount);
    }
```

Listing 3.8: Meta::mint()

Also, the owner of the CashierDesk protocol takes the important responsibility to manage callers, who are able to reduce the balance of the tokens deposited by user into this contract.

```
    function SubUsersBalance(
        address token,
        address[] memory users,
        uint256[] memory values
    ) public onlyCaller returns (bool) {
        require(_support_token.contains(token) == true, "cant support token.");
        require(users.length == values.length, "bad length");

        for (uint256 i = 0; i < users.length; i++) {
            require(
                _balanceOf[users[i]][token] >= values[i],
                "enougth amount."
            );

            _balanceOf[users[i]][token] -= values[i];
```

```
            emit SubToken(users[i], token, values[i], block.timestamp);
        }

        return true;
    }
```

Listing 3.9: `CashierDesk::SubUsersBalance()`

It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that they will gradually switch to DAO in the future.

3.6 Proper Handling of Unsupported Token in RewardTheAuthor::claim()

- ID: PVE-006
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `RewardTheAuthor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, the `RewardTheAuthor` contract provides a `claim()` routine for the author to claim the rewards from the user. To elaborate, we show below the related routines.

```
    function claim(address token) public {
        uint256 pending = _userRewards[msgSender()][token];
        if (pending == 0) return;

        _userRewards[msgSender()][token] = 0;
        _userClaimedRewards[msgSender()][address(token)] = _userClaimedRewards[msgSender
            ()][address(token)].add(pending);

        IERC20(token).safeTransfer(msgSender(), pending);
```

```

        emit Claim(msgSender(), token, pending);
    }
```

Listing 3.10: `RewardTheAuthor::claim()`

```
    function addSupportToken(address token) public onlyOwner {
        require(token != address(0), "address is zero");
        require(!_supportTokens.contains(token), "already added");

        _supportTokens.add(token);
    }
```

Listing 3.11: `RewardTheAuthor::addSupportToken()`

```
    function delSupportToken(address token) public onlyOwner {
        require(_supportTokens.contains(token), "not added");

        _supportTokens.remove(token);
    }
```

Listing 3.12: `RewardTheAuthor::delSupportToken()`

We notice the token deposited into the contract via `reward()` could be deleted from the `_supportTokens` by the `owner`. However, there is no constrain to prohibit the author from withdrawing the unsupported from the contract.

Recommendation Proper handling of unsupported token in `RewardTheAuthor::claim()`.

Status `75d27f8`.

4 Conclusion

In this audit, we have analyzed the `Metatime` protocol design and implementation. `Metatime` is a `SocialFi` project built on `Binance Smart Chain (BSC)`. It is a `metaverse` project based on `Time` and `Matter` aiming to encourage people to cherish time and create value for their time. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.