The Materials to Select Reentrancy Related Vulnerabilities

To correctly select the reentrancy related vulnerabilities, we first refer to the descriptions in the original papers or GitHub repositories to show that our selected tools are able to detect reentrancy vulnerabilities. Then, we refer to the implementation code or documentation of these tools to show all the bug categories with descriptions supported by these tools. The reentrancy-related descriptions are highlighted in the corresponding categories.

Note: Mythril and Securify(V2) have mapped their bug categories to SWC-ID in the SWC registry (https://swcregistry.io/), which summarizes 37 well-known smart contract weaknesses. SWC-107 (https://swcregistry.io/docs/SWC-107) is the reentrancy bug with the following description:

"*One of the major dangers of calling external contracts is that they can take over the control flow. In the* ***reentrancy attack*** *(a.k.a. recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.*"

1. Oyente

The original paper of Oyente:
https://www.comp.nus.edu.sg/~prateeks/papers/Oyente.pdf

The GitHub repository of Oyente: https://github.com/enzymefinance/oyente

Oyente is one of the first smart contract vulnerability analyzers and is able to detect reentrancy vulnerability and three other smart contract bugs, as shown in Fig. 1.

- *TOD detection.* Explorer returns a set of traces and the corresponding Ether flow for each trace. Our analysis thus checks if two different traces have different Ether flows. If a contract has such pairs of traces, OYENTE reports it as a TOD contract.
- *Timestamp dependence detection.* We use a special symbolic variable to represent the block timestamp. Note that the block timestamp stays constant during the execution. Thus, given a path condition of a trace, we check if this symbolic variable is included. A contract is flagged as timestamp-dependent if any of its traces depends on this symbolic variable.
- *Mishandled exceptions.* Detecting a mishandled exception is straightforward. Recall that if a callee yields an exception, it pushes 0 to the caller's operand stack. Thus we only need to check if the contract executes the ISZERO instruction (which checks if the top value of the stack is 0) after every call. If it does not, any exception occurred in the callee is ignored. Thus, we flags such contract as a contract that mishandles exceptions.
- *Reentrancy Detection.* We make use of path conditions in order to check for reentrancy vulnerability. At each CALL that is encountered, we obtain the path condition for the execution before the CALL is executed. We then check if such condition with updated variables (*e.g.*, storage values) still holds (*i.e.*, if the call can be executed again). If so, we consider this a vulnerability, since it is possible for the callee to re-execute the call before finishing it.

Figure 1: Oyente analyzed reentrancy vulnerability in their paper

In the current implementation of Oyente, the authors extended the vulnerabilities they can detect, including Reentrancy and four other categories, as shown in Fig. 2. The code can be found at:
https://github.com/enzymefinance/oyente/blob/master/web/app/helpers/source_code_helper.rb

```
def vulnerability_names
  return {
    callstack: "Callstack Depth Attack Vulnerability",
    time_dependency: "Timestamp Dependency",
    reentrancy: "Re-Entrancy Vulnerability",
    money_concurrency: "Transaction-Ordering Dependence (TOD)",
    assertion_failure: "Assertion Failure"
  }
end
```

Figure 2: All vulnerabilities supported by Oyente implementation

2. Mythril

The documentation of Mythril on the GitHub repository:
https://mythril-classic.readthedocs.io/en/master/module-list.html

Mythril is an industrial analyzer proposed by ConsenSys and has been regularly updated over the years. It uses two analysis modules to detect SWC-107 (Reentrancy).

1) The first analysis module for Reentrancy was the "**External Calls**" module:

External Calls

The external calls module warns about SWC-107 (Reentrancy) by detecting calls to external contracts.

Figure 3: The description of the External Calls Module in Mythril

This module raises the category "**External Call To User-Supplied Address**" if Reentrancy is detected:
https://github.com/ConsenSys/mythril/blob/develop/mythril/analysis/module/modules/external_calls.py

```
            issue = PotentialIssue(
                contract=state.environment.active_account.contract_name,
                function_name=state.environment.active_function_name,
                address=address,
                swc_id=REENTRANCY,
                title="External Call To User-Supplied Address",
```

Figure 4: Code snippet in External Calls module

2) The second analysis module for Reentrancy is the "**State Change External Calls**" module:

State Change External Calls

The state change external calls module detects SWC-107 (Reentrancy) by detecting state change after calls to an external contract.

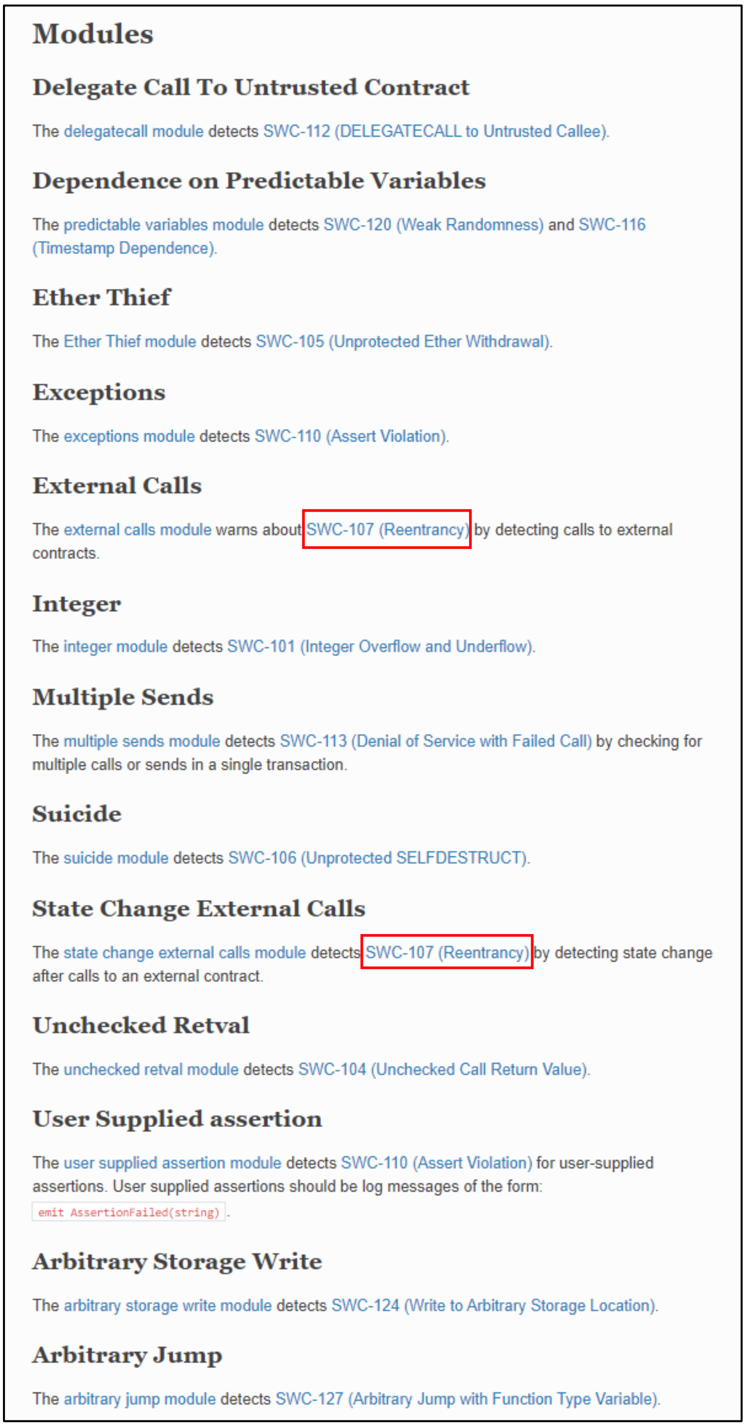
Figure 5: The description of the State Change External Calls Module in Mythril

This module raises the category "**State access after external call**" if Reentrancy is detected:
https://github.com/ConsenSys/mythril/blob/develop/mythril/analysis/module/modules/state_change_external_calls.py

```
        return PotentialIssue(
            contract=global_state.environment.active_account.contract_name,
            function_name=global_state.environment.active_function_name,
            address=address,
            title="State access after external call",
```

Figure 6: Code snippet in State Change External Calls module

Mythril also provides other modules to detect other categories of vulnerabilities. All the analysis modules could be found at: https://mythril-classic.readthedocs.io/en/master/module-list.html, as listed in Fig. 7.



Modules

Delegate Call To Untrusted Contract

The delegatecall module detects SWC-112 (DELEGATECALL to Untrusted Callee).

Dependence on Predictable Variables

The predictable variables module detects SWC-120 (Weak Randomness) and SWC-116 (Timestamp Dependence).

Ether Thief

The Ether Thief module detects SWC-105 (Unprotected Ether Withdrawal).

Exceptions

The exceptions module detects SWC-110 (Assert Violation).

External Calls

The external calls module warns about SWC-107 (Reentrancy) by detecting calls to external contracts.

Integer

The integer module detects SWC-101 (Integer Overflow and Underflow).

Multiple Sends

The multiple sends module detects SWC-113 (Denial of Service with Failed Call) by checking for multiple calls or sends in a single transaction.

Suicide

The suicide module detects SWC-106 (Unprotected SELFDESTRUCT).

State Change External Calls

The state change external calls module detects SWC-107 (Reentrancy) by detecting state change after calls to an external contract.

Unchecked Retval

The unchecked retval module detects SWC-104 (Unchecked Call Return Value).

User Supplied assertion

The user supplied assertion module detects SWC-110 (Assert Violation) for user-supplied assertions. User supplied assertions should be log messages of the form: `emit AssertionFailed(string)`.

Arbitrary Storage Write

The arbitrary storage write module detects SWC-124 (Write to Arbitrary Storage Location).

Arbitrary Jump

The arbitrary jump module detects SWC-127 (Arbitrary Jump with Function Type Variable).

Figure 7: All the analysis modules with descriptions in Mythril

3. Securify(V1)

The original paper of Securify(V1):
https://files.sri.inf.ethz.ch/website/papers/ccs18-securify.pdf

The documentation of Securify(V1) on the GitHub repository:
https://github.com/eth-sri/securify

In the original paper, Securify was compared with Oyente and Mythril on the detection of several vulnerabilities including Reentrancy, as shown in Fig. 8.

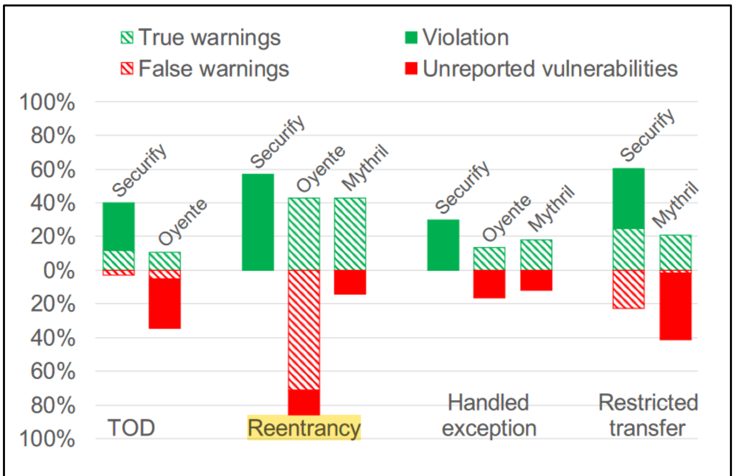


Figure 13: Comparing SECURIFY to Oyente and Mythril.

Figure 8: The experiment results in the original paper of Securify.

Meanwhile, Securify is implemented as two versions: Securify(V1) and Securify(V2). From the implementation code of Securify(V1), we find that Securify(V1) has two categories related to reentrancy (SWC-107).

1) The first category is "**DAO**". The link to the related code is: https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/DAO.java

```
public class DAO extends AbstractInstructionPattern {

    public DAO() {
        super(new PatternDescription("RecursiveCalls",
                DAO.class,
                "Gas-dependent Reentrancy",
                "Calls into external contracts that receive all remaining gas and are followed by state changes may be reentrant.",
                PatternDescription.Severity.Critical,
                PatternDescription.Type.Security));
    }
```

2) The second category is "**DAOConstantGas**". The link to the related code is https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/DAOConstantGas.java

```
public class DAOConstantGas extends AbstractInstructionPattern {

    public DAOConstantGas() {
        super(new PatternDescription("RecursiveCalls",
                DAOConstantGas.class,
                "Reentrancy with constant gas",
                "Ether transfers (such as send and transfer) that are followed by state changes may be reentrant.",
                PatternDescription.Severity.Critical,
                PatternDescription.Type.Security));
    }
```

Securify(V1) also provides other categories, which can be found at: https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/

We list these categories in Table 2 for simplicity.

Table 2. All vulnerability categories supported by Securify(V1)

Vulnerability	Description
DAO	**Calls into external contracts that receive all remaining gas and are followed by state changes may be reentrant.**
DAOConstantGas	**Ether transfers (such as send and transfer) that are followed by state changes may be reentrant.**
Missing Input Validation	Method arguments must be sanitized before they are used in computation.
TODTransfer	Ether transfers whose execution can be manipulated by other transactions must be inspected for unintended behavior.
TODReceiver	The receiver of ether transfers must not be influenced by other transactions.
TODAmount	The amount of ether transferred must not be influenced by other transactions.
Unhandled Exception	The return value of statements that may return error values must be explicitly checked.
UnrestrictedEtherFlow	The execution of ether flows should be restricted to an authorized set of users.

a) Securify(V2)

The original paper of Securify(V2):
https://files.sri.inf.ethz.ch/website/papers/ccs18-securify.pdf

The documentation of Securify(V2) on the GitHub repository:
https://github.com/eth-sri/securify2

In the implementation of Securify(V2), we find that Securify(V2) has four categories related to reentrancy (SWC-107).

1) The first category is "**Benign Reentrancy**". The output is as follows:

```
Pattern:     Benign Reentrancy
Description: Reentrancy is equivalent with two consecutive calls of
             the function
```

2) The second category is "**Reentrancy with constant gas**". The output is as follows:

```
Pattern:     Reentrancy with constant gas
Description: Ether transfers (such as send and transfer) that are
             followed by state changes may be reentrant.
```

3) The third category is "**Gas-dependent Reentrancy**". The output is as follows:

```
Pattern:     Gas-dependent Reentrancy
Description: Calls into external contracts that receive all remaining
             gas and are followed by state changes may be reentrant.
```

4) The fourth category is "**No-Ether-Involved Reentrancy**". The output is as follows:

```
Pattern:     No-Ether-Involved Reentrancy
Description: Reentrancy that involves no ether
```

Securify(V2) also supports other categories of vulnerabilities. Similarly, we listed all the categories supported by Securify(V2) in Table 3. The information could also be found at: https://github.com/eth-sri/securify2 and https://github.com/eth-sri/securify2/tree/master/securify/staticanalysis/souffle_analysis/patterns.

Table 3. All vulnerability categories supported by Securify(V2)

Vulnerability	**Description**	**SWC-ID**
Reentrancy with constant gas	**Ether transfers (such as send and transfer) that are followed by state changes may be reentrant.**	**SWC-107**
Benign Reentrancy	**Reentrancy is equivalent with two consecutive calls of the functions.**	**SWC-107**
Gas-dependent Reentrancy	**Calls into external contracts that receive all remaining gas and are followed by state changes may be reentrant.**	**SWC-107**
No-Ether-Involved Reentrancy	**Reentrancy that involves no ether.**	**SWC-107**
Assembly Usage	Usage of assembly in Solidity code is discouraged	-
Constable State Variables	State variables that do not change should be declared as constants.	-
External Calls of Functions	A public function that is never called within the contract should be marked as external	-
Low Level Calls	Usage of <address>.call should be avoided	-
Missing Input Validation	Method arguments must be sanitized before they are used in computation.	-
Solidity Naming Convention	Reports declarations that do not adhere to Solidity's naming convention.	-
State variables default visibility	Visibility of state variables should be stated explicitly.	SWC-108
Unhandled Exception	The return value of statements that may return error values must be explicitly checked.	-
Uninitialized Local Variables	A variable is declared but never initialized.	SWC-109
Uninitialized State Variable	State variables should be explicitly initialized.	SWC-109
Unrestricted write to storage	Contract fields that can be modified by any user must be inspected.	SWC-124

Unused Return Pattern	The value returned by an external function call is never used.	SWC-104
Usage of block timestamp	Returned value relies on block timestamp.	SWC-116
Solidity pragma directives	Avoid complex solidity version pragma statements.	SWC-103
Too Many Digit Literals	Usage of assembly in Solidity code is discouraged.	-
Transaction Order Affects Ether Amount	The amount of ether transferred must not be influenced by other transactions.	SWC-114
Transaction Order Affects Ether Receiver	The receiver of ether transfers must not be influenced by other transactions.	SWC-114
Transaction Order Affects Execution of Ether Transfer	Ether transfers whose execution can be manipulated by other transactions must be inspected for unintended behavior.	SWC-114
Unrestricted Ether Flow	The execution of ether flows should be restricted to an authorized set of users.	SWC-105
Multiplication after division	Information might be lost due to division before multiplication.	-
Shadowed Local Variable	Reports local variable declarations that shadow declarations from outer scopes.	-
Locked Ether	Contracts that may receive ether must also allow users to extract the deposited ether from the contract.	-
ERC20 Indexed Pattern	Events defined by ERC20 specification should use the indexed' keyword.	-
External call in loop	If a single call in the loop fails or revers, it will cause all other calls to fail as well.	SWC-104
Unused State Variable	Unused state variables should be removed.	-
Repeated Call to Untrusted Contract	Repeated call to an untrusted contract may result in different values.	-
Dangerous Strict Equalities	Strict equalities that use account's balance, timestamps and block numbers should be avoided.	-
Shadowed Builtin	Reports declarations that shadow Solidity's builtin symbols.	-
State Variable Shadowing	State variables in inherited contract should not be named identically to inherited variables.	SWC-119
Possibly unsafe usage of tx-origin	The return value of statements that may return error values must be explicitly checked.	SWC-115
Unrestricted call to selfdestruct	Calls to selfdestruct that can be triggered by any user must be inspected.	SWC-106
Call to Default Constructor	A call to the constructor might be a call to a normal function instead.	-

Delegatecall or callcode to unrestricted address	The address of a delegatecall or callcode must be approved by the contract owner.	SWC-112

4. Smartian

The original paper of Smartian: https://agroce.github.io/ase21.pdf

The GitHub repository of Smartian: https://github.com/SoftSec-KAIST/Smartian

Smartian supports 13 bug categories including reentrancy as follows.

ID	Bug Name	Description
AF	Assertion Failure	The condition of an `assert` statement is not satisfied [2].
AW	Arbitrary Write	An attacker can overwrite arbitrary storage data by accessing a mismanaged array object [12].
BD	Block State Dependency	Block states (e.g. timestamp, number) decide ether transfer of a contract [36], [44].
CH	Control-flow Hijack	An attacker can arbitrarily control the destination of a `JUMP` or `DELEGATECALL` instruction [1], [36].
EL	Ether Leak	A contract allows an arbitrary user to freely retrieve ether from the contract [54].
FE	Freezing Ether†	A contract can receive ether but does not have any means to send out ether [36], [54].
IB	Integer Bug	Integer overflows or underflows occur, and the result becomes an unexpected value.
ME	Mishandled Exception	A contract does not check for an exception when calling external functions or sending ether [36], [44].
MS	Multiple Send	A contract sends out ether multiple times within one transaction. This is a specific case of DoS [5].
RE	Reentrancy	A function in a victim contract is re-entered and leads to a race condition on state variables [44].
RV	Requirement Violation‡	The condition of a `require` statement is not satisfied [8].
SC	Suicidal Contract	An arbitrary user can destroy a victim contract by running a `SELFDESTRUCT` instruction [54].
TO	Tranasaction Origin Use	A contract relies on the origin of a transaction (i.e. `tx.origin`) for user authorization [3].

Figure 9: Bug categories supported by Smartian

Meanwhile, Smartian compared their tool with four other tools on TP and FP rates of several vulnerabilities including Reentrancy, as shown in Fig. 10.

Bug ID	SMARTIAN		ILF		sFuzz		Manticore		Mythril	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
BD	11	0	0	0	10	0	6	5	8	0
ME	48	0	10	0	29	6	18	0	46	0
RE	19	0	15	2	5	20	19	3	19	38

Figure 10: Number of TP and FP alarms raised by each tool on benchmark

In the implementation, Smartian raises bug categories including "**Reentrancy**" when it found bugs in smart contracts. The related code can be found at: https://github.com/SoftSec-KAIST/Smartian/blob/main/src/Fuzz/TCManage.fs

```
log "Found Bugs:"
log "  Assertion Failure: %d" totalAF
log "  Arbitrary Write: %d" totalAW
log "  Block state Dependency: %d" totalBD
log "  Control Hijack: %d" totalCH
log "  Ether Leak: %d" totalEL
log "  Integer Bug: %d" totalIB
log "  Mishandled Exception: %d" totalME
log "  Multiple Send: %d" totalMS
log "  Reentrancy: %d" totalRE
log "  Suicidal Contract: %d" totalSC
log "  Transaction Origin Use: %d" totalTO
log "  Freezing Ether: %d" totalFE
log "  Requirement Violation: %d" totalRV
```

Figure 11: Code Snippet indicating all the bug categories supported by Smartian

5. Sailfish

The original paper of Sailfish: https://arxiv.org/pdf/2104.08638.pdf

The GitHub repository of Sailfish: https://github.com/ucsb-seclab/sailfish

Sailfish formally defines two vulnerability patterns in their paper, i.e., **reentrancy** bug and **TOD** bug, as shown in Fig. 12.

Definition 5 (Reentrancy bug). If a contract $\mathcal{C}$ contains an SI bug due to two schedules $\mathcal{H}_1$ and $\mathcal{H}_2 = \mu(\mathcal{H}_1)$, such that $\exists e \in \mathcal{H}_2\ (e.pc \neq 0)$ (first transformation strategy), then the contract is said to have a reentrancy bug.

Definition 6 (Generalized TOD bug). If a contract $\mathcal{C}$ contains an SI bug due to two schedules $\mathcal{H}_1$ and $\mathcal{H}_2 = \mu(\mathcal{H}_1)$, such that $\mathcal{H}_2$ is a permutation (second transformation strategy) of $\mathcal{H}_1$, then the contract is said to have a generalized transaction order dependence (G-TOD), or event ordering bug (EO) [43].

Figure 12: Two bug categories supported by Sailfish

In the implementation, Sailfish supports two types of vulnerabilities, which are named "DAO" and "TOD". The "TOD" category corresponds to the TOD vulnerability; and

the “DAO” category corresponds to the Reentrancy vulnerability. The related code can be found at:
https://github.com/ucsb-seclab/sailfish/blob/master/code/static_analysis/analysis/detection.py#L65.

```
    def setup(self):
        self.global_vars, self.global_constant_vars, self.range_vars, self.global_blocks, self.range_blocks, self.total_range_instructions = compute_range_blocks(self._slither, self.vrg_obj, self._log)

        if self.dao is True:
            self.detect_dao_patterns()
        if self.tod is True:
            self.detect_tod_patterns()
```

Figure 13: Code snippet indicating all bug categories supported by Sailfish