Vyper: A Security Comparison with Solidity Based on Common Vulnerabilities

Mudabbir Kaleem
University of Houston
Houston, Texas

Anastasia Mavridou
KBR / NASA Ames Research Center
Mountain View, California

Aron Laszka
University of Houston
Houston, Texas

Abstract—**Vyper has been proposed as a new high-level language for Ethereum smart contract development due to numerous security vulnerabilities and attacks witnessed on contracts written in Solidity since the system's inception. Vyper aims to address these vulnerabilities by providing a language that focuses on simplicity, auditability and security. We present a survey where we study how well-known and commonly-encountered vulnerabilities in Solidity feature in Vyper's development environment. We analyze all such vulnerabilities individually and classify them into five groups based on their status in Vyper. To the best of our knowledge, our survey is the first attempt to study security vulnerabilities in Vyper.**

I. INTRODUCTION

Ethereum [1] is an open source, blockchain-based distributed computing platform that features smart contract [2] functionality. A smart contract is essentially a set of rules and procedures enforced digitally through pieces of Turing-complete code. Ethereum went live in July 2015, and is currently the most popular blockchain-based system for deploying smart contracts. Ethereum stores smart contracts in a public distributed database, i.e., a blockchain [3]. Smart contracts are executed by the participants of the network via transactions. Smart contracts exist on the Ethereum blockchain in opcode form. These opcodes are executed by the nodes of the Ethereum network inside the Ethereum Virtual Machine [4] which is a stack based virtual execution environment. Having contracts executed inside the EVM ensures that the execution results are deterministic and identical in all the nodes regardless of their underlying computing capabilities.

Almost since the inception of Ethereum, Solidity [5] has been the most popular high level language for writing smart contracts. Developed by the contributors of the Ethereum project, Solidity continues to be the most popular tool for writing Ethereum smart contracts today. However, smart contracts written in Solidity are riddled with security vulnerabilities, which have been exploited in many highly publicized attacks on various Ethereum based projects [6]. Table I lists some of the most notable incidents due to smart-contract vulnerabilities. Although Solidity has undergone many revisions and updates in order to address these vulnerabilities, multiple improvements can still be made. Perhaps the major reason of

Solidity-written smart contracts encountering vulnerabilities, is that the language has been influenced by JavaScript and C++. Uninitiated developers, coming from web development and other backgrounds are lead to draw parallels between Solidity and these languages even where they do not exist. To be able to write secure smart contracts in Solidity, it is essential to have a sound understanding of the underlying Ethereum system and its various subtleties.

Recently, an alternate to Solidity, Vyper [7] has been developed to offer a better medium for writing smart contracts that are easier to understand. Vyper aims to make it harder for developers to intentionally write misleading or malicious code and also protects developers from unintendedly leaving vulnerabilities in their contract code. A stable version of Vyper is yet to be released but we offer a comparison between Vyper's latest beta version at the time of writing, i.e., 0.1.0-beta.15 and Solidity's latest release at the time of writing, i.e., v0.6.2. It is worth mentioning that Vyper does not claim to be a replacement for Solidity. It actually claims to strive towards goals of auditability, simplicity and security. To achieve these goals, it sacrifices various features and functionalities found in Solidity. This is done to make code written in Vyper more human readable, especially for the inexperienced users. If any of the more complex features of Solidity, which Vyper does not adapt, is required by the programmer then they will have to use Solidity for writing the smart contract. Moreover, Vyper also introduces additional features to support security and readability.

This paper presents a comparison of how various vulnerabilities, which are known to exist in the domain of smart contract development in Solidity, feature in Vyper's environment. The paper targets smart contract developers, users, and researchers who can use this resource to get up to speed with Vyper's current standing on known security issues. We are not currently aware of any other literature that provides such a comparison or details Vyper's vulnerabilities. In Section II, the paper first outlines Vyper's principles and design goals along with the various features added in Vyper to improve upon Solidity. It also enumerates the features that exist in Solidity but have not been provided in Vyper in order to achieve its design goals. In Section III, we present a taxonomy of the known vulnerabilities in smart contract development in Solidity and compare how Vyper features in each of those vulnerabilities.

Incident	Date	Amount	Vulnerabilities	Mitigation	References
King of Ether Throne	February 6–8, 2016	98 Ether	Insufficient gas send, Exception for external call not handled	Manually sending back the failed transactions to participants	[8]
Rubixi Vulnerability	April 2016	1.3k Ether	Wrong constructor name		[9]
GovernMental	April 2016	1.3k Ether	Insufficient gas	$\sim$50 ETH transaction fee paid to raise gas limit	[9]
The DAO Attack	June 16, 2016	3.6M Ether	Reentrancy	Addressed with a fork to the Ethereum blockchain	[10], [11]
Parity Wallet Hack	July 19, 2017	150k Ether	Missing access control		[12]
Parity Wallet Freeze	November 6, 2017	$\sim$500k Ether	Unprotected suicide		[13]
POWH Coin Hack	January 28, 2018	2k Ether	Integer overflow		[14]
BEC Token Attack	April 2018		Integer overflow		[15]
Fomo3D Attack	August 22, 2018	10.5k Ether	Block stuffing		[16]
SpankChain Attack	October 8, 2018	170 Ether	Reentrancy		[17]

II. VYPER PRINCIPLES AND FEATURES

Vyper, according to its official documentation, is a contract-oriented, pythonic programming language targeting the Ethereum virtual machine. Although Vyper is still in development and a production ready release is yet to come out, we base this paper on the latest beta release, i.e., v0.1.0-beta.15. We do not expect there to be major design changes once a stable version comes out or once the Vyper project migrates from its current Python based compiler to a Rust based compiler [18]. Vyper claims to be designed towards achieving the following three design goals or principles [7]:

- **Language and compiler simplicity:** Vyper aims to keep the language and the compiler implementation as simple as possible.
- **Security:** Vyper aims to provide the programmer the ability to write smart contracts without any undesired vulnerabilities or loopholes.
- **Auditability:** Vyper is aimed at making smart contracts easy to read for the users, especially those with insignificant prior experience with smart contracts or programming in general. Users should be able to identify malicious contracts with minimal effort. Vyper claims to give user readability preference over even the development experience for writing the contracts.

In order to achieve these desired goals, Vyper provides the following features not found in Solidity [7]:

1) Bounds and overflow checking on array accesses and arithmetic;
2) Support for signed integers and decimal fixed point numbers;
3) Decidability: possible to compute the precise upper bound for the gas consumption of any Vyper function.;
4) Strong typing, including support for units (e.g., timestamp, timedelta, seconds, wei, wei per second, meters per second squared);
5) Small and understandable compiler code;
6) Limited support for pure functions: anything marked constant is not allowed to change the state..

Vyper also does not contain many of the features found in Solidity in order to achieve its desired objectives of minimal complexity and easy-to-do auditability by inexperienced developers. Following are the features that it does not contain:

1) Binary Fixed Point: to avoid approximations associated with using binary fixed point.
2) Recursive Calling: makes it impossible to compute an upper bound on gas consumption.
3) Operator Overloading: makes writing misleading or complex code possible.
4) Class Inheritance: makes understanding the code complex since precedence rules come into play in case of conflicts.
5) Inline Assembly: makes it impossible to search around for all instances of a variable name.
6) Function Overloading: Can be confusing for the inexperienced programmer to keep track of which instance is executed
7) Infinite-Length Loops: makes it impossible to compute an upper bound on gas consumption.
8) Modifiers: makes it easy to write misleading code.

III. COMPARISON OF VYPER WITH SOLIDITY'S VULNERABILITIES

In this section we provide a detailed taxonomy of commonly known vulnerabilities in Solidity smart contracts and compare how each vulnerability fares in Vyper. Although numerous resources list the known vulnerabilities and attacks on smart contracts developed in Solidity, we use the vulnerabilities listed in Chen et al. [6] as our base reference since we believe that this paper is the most comprehensive publication of these known vulnerabilities. Chen et al. attribute 19 vulnerabilities in Ethereum systems security either to smart contract programming or to the Solidity language and toolchain. We analyze each of these vulnerabilities in Vyper's context and present our findings. Vyper may introduce additional vulnerabilities of its own in smart contract development that have not been observed in Solidity. However, given that the Vyper project is still in development and due to inadequate resources and test cases we do not attempt to study those in the present survey.

Vulnerabilities in Solidity	FA	PA	NA	AA	NP
Integer overflow and underflow	✗				
DoS with unbounded operation	✗				
Unchecked call return value	✗				
Reentrancy		✗			
Delegatecall injection			✗		
Forced Ether to contract			✗		
DoS with unexpected revert			✗		
Erroneous visibility				✗	
Uninitialized storage pointer				✗	
Erroneous constructor name				✗	
Upgradeable contract					✗
Type casts					✗
Insufficient signature information					✗
Frozen Ether					✗
Authentication through `tx.origin`					✗
Unprotected suicide					✗
Leaking Ether to arbitrary address					✗
Secrecy failure					✗
Outdated compiler version					✗

This survey is intended to only explore how the currently known vulnerabilities translate to Vyper's environment.

We divide these 19 known vulnerabilities into five groups (Table II) which are: 1) vulnerabilities addressed by Vyper, 2) vulnerabilities partially addressed by Vyper, 3) vulnerabilities not addressed by Vyper, 4) vulnerabilities that have already been mitigated in Solidity and 5) vulnerabilities which we believe are not addressable by the programming language or the tool chain, despite being listed as such by Chen et al.

The first group consists of currently existing vulnerabilities in Solidity that have been addressed in Vyper by providing an additional function or feature or disallowing specific features. These vulnerabilities can be completely avoided in Vyper. The second group consists of vulnerabilities which have been partially addressed by Vyper but still may exist if proper development practices are not followed by the developer. The third group consists of vulnerabilities that still exist both in Solidity and Vyper if the best programming practices and recommendations are not followed. The fourth group consists of historical vulnerabilities in the Solidity environment that were mitigated through later Solidity releases and do not exist in Solidity anymore and are also not present in Vyper. The fifth group consists of those vulnerabilities that have been listed by Chen et al [6] as being caused by smart contract programming or the underlying Solidity language and toolchain. However, in the context of this survey, we argue that these vulnerabilities can only be avoided through proper understanding of the underlying Ethereum system on part of the programmer and following the best programming practices and security recommendations. For this group of vulnerabilities, Vyper or any other high level language is not a candidate to address them. However, these vulnerabilities may be addressed by

design and verification tools for smart contracts [19], [20], [21], [22], [23], [24], [25]. For detailed discussion of such tools, we refer the reader to relevant surveys [26], [27], [6], [28], [29], [30]. We now proceed to describe each of these 19 vulnerabilities in Vyper's context and provide reasoning for the classification of each of these into their respective group.

A. Vulnerabilities addressed by Vyper

1) Integer overflow and underflow: This vulnerability occurs due to the fact that both Solidity and the EVM do not enforce integer overflow / underflow detection. This can lead to attacks which make unauthorized or unintended manipulation to a contract's state variables if proper measures were not taken during development. Libraries such as SafeMath [31] do provide mechanisms for protecting against over/underflows in Solidity but Vyper has this feature built-in. In Vyper, the contract execution will revert if an over/underflow is detected [32].

2) DoS with unbounded operations: This vulnerability occurs when the operations required in the execution of a function exceed the block gas limit due to unbounded operations either in the contract itself or in one of the called contracts. Vyper solves this problem by having a precise upper bound for the gas consumption of any function call. This is possible because infinite length loops and recursive function calling are not allowed in Vyper [7].

3) Unchecked call return value: This vulnerability exists due to the discrepancy in Solidity's handling of exceptions occurring in callee contracts. Solidity handles exceptions when calling another contract in two ways: (1) when directly referencing the callee's contract instance or using the `transfer()` function; (2) when using one of the four low level methods (`call`, `staticcall`, `delegatecall`, and `send`). In the first instance, the exception is "bubbled up" and the entire transaction is reverted whereas in the second case only a `false` is returned to the calling contract. The uninitiated developer can be misled to think that any call(s) to other contracts were successful because no exception was thrown in the latter case. Solidity does not enforce any checks on the return values. In comparison, Vyper only provides two ways to call another contract in addition to the direct reference, i.e., the functions `send()` and `raw_call()` [33]. The current Vyper compiler has built-in asserts for both of these functions [34], so that in case of a failure the entire transaction will be reverted.

B. Vulnerabilities partially addressed by Vyper

1) Reentrancy: The reentrancy vulnerability occurs when a contract calls an external contract, handing it over the execution control, which allows the callee to call back to the calling contract and then be able to perform some malicious steps. A contract is particularly vulnerable to reentrancy attacks if it does not make the necessary state changes before calling the external contract or if the code does not protect against multi-contract access situations. Vyper provides the functionality to the programmer to protect a contract against multi-contract

access situations by providing a `nonreentrant` decorator which places a lock on the current function and all functions with the same key value [35]. Fig. 1 provides an example of how to use this feature (for function `sendFunc`). If any external callee tries to callback into such functions, it will result in a revert call. Solidity did not provide such functionality and the developer had to implement locks or mutexes themselves or through some third party libraries [36]. However, even in Vyper, the developer still has to identify the functions or blocks of code that might be susceptible to such a vulnerability and also has to ensure that all necessary state changes are made before making an external interaction. The current Vyper compiler does not warn the developer for such cases.

```
1  sent: public(bool)
2
3  @public
4  def __init__():
5      self.sent = False
6
7  @public
8  @nonreentrant("exampleKey")
9  def sendFunc(to: address) -> uint256:
10     if self.sent == False:
11         send(to, 1)
12         self.sent = True
13     return 1
```

Fig. 1. Nonreentrant decorator feature.

C. Vulnerabilities not addressed by Vyper

1) Delegatecall Injection: EVM provides the option of calling an external contract with the context of the caller contract using the `DELEGATECALL` opcode. This is achieved by using the `delegatecall` function in Solidity and using the `raw_call` function with the `delegate_call` keyword argument set to `True` in Vyper, e.g., `raw_call(argAddress, example_bytes, outsize=0, gas=10000, value=1, delegate_call=True)`. However, if the contract being called is malicious, it can manipulate the state variables of caller contract. This vulnerability can be mitigated in Solidity by only using the `DELEGATECALL` with contracts that have been declared as libraries. In Vyper this vulnerability can similarly be avoided by only using `DELEGATECALL` with functions that are declared with the `@constant` decorator, which ensures that the functions will not mutate the state. However, like Solidity, this is not enforced in Vyper because there are legitimate cases, using the `DELEGATECALL` opcode, where the caller wants the callee to modify its state. Perhaps the best way to completely avoid this vulnerability is for the EVM to provide another opcode (just like `DELEGATECALL`) which retains the context of the caller contract but causes a revert if the callee tries to make any changes to the caller's state (just like `STATICCALL`). Another possible solution at the language level could be to have the compiler place appropriate checks on state variables before and after the `DELEGATECALL` opcode is used and to give the user an option to enable to disable these checks.

2) Forced Ether to contract: This vulnerability occurs when the developer of the smart contract incorrectly assumes that the contracts fallback or payable function will be executed each time Ether is transferred to the contract. There are two situations in which Ether can be sent to a contract without invoking its fallback or payable functions. Firstly, when a contract that is self-destructing sends its remaining Ether to the contract or secondly, if Ether is transferred to an address even before the contract is loaded to that address. The second situation is possible because the contract addresses are deterministic and can be calculated before deploying them [37]. This vulnerability is due to the design of the underlying Ethereum protocol and the developer has to be aware of Ethereum's design and functionality when writing smart contracts. This vulnerability can be avoided if the contract does not place checks on the exact values of the contract's balance (`self.balance`). Currently, the Vyper compiler does not warn the developer if checks are placed on the `self.balance` variable in the contract code.

Another possible workaround could be to have a built-in mechanism in contracts to run the payable or fallback functions if a contract is invoked and its balance is different from its last invocation (meaning Ether was forced to the contract in between the invocations).

3) DoS with unexpected revert: This vulnerability occurs when an external contract causes a revert resulting in disruption of execution of the caller contract before it has completed its function [38]. The most common scenario is when the developer fails to account for the case when a payment is made to an external contract whose fallback or payable function execution results in a revert. This vulnerability is addressed in Solidity and Vyper by making use of a pull rather that push based mechanism when making external payments [39]. Alternatively, contracts in Solidity can take measures to handle the cases in code where an external call might throw an exception. Vyper currently, does not allow the handling of exceptions.

D. Vulnerabilities addressed in Solidity or Ethereum

1) Erroneous Visibility: This vulnerability occurs when a contract's visibility is incorrectly specified and thus permits unauthorized access. Solidity used to make functions public by default if the visibility was not specified. However, this was addressed with version 0.5.0 by making it compulsory to specify visibility when defining functions [40]. Vyper allows functions without visibility (v0.1.0beta15) but defaults them to being of private visibility instead of public.

2) Uninitialized storage pointer: This vulnerability occurs due to the fact that prior to Solidity 0.5.0, if a complicated local variable (e.g., struct, array or mapping) was not explicitly initialized at the time of declaration, then the local variable's reference points to slot 0 in storage by default, possibly overwriting a state variable [41]. Since Solidity v0.5.0, the Solidity compiler reports an error to contracts that contain

uninitialized storage pointers. Also, explicit data location (i.e., storage, memory, or calldata) for all variables of struct, array or mapping type is now mandatory in Solidity [40]. Vyper also mandates the initialization of local variables at the time of declaration and failure to do so results in a compile time error.

3) Erroneous constructor name: This vulnerability occurred due to the fact that prior to Solidity version 0.4.22, a function declared with the same name as the contract was considered to be the contract's constructor function. A constructor function is called only once at the time of contract creation to perform initialization. If the programmer accidentally misspelled this function name then it became a public function which allowed anyone to call it and possibly compromise the contract [42]. This vulnerability was mitigated in Solidity version 0.4.22 by introducing the usage of mandatory keyword `constructor` when defining the constructor function [43]. Similarly, Vyper uses the keyword `__init__` for the constructor.

E. Vulnerabilities not addressable by language or toolchain

1) Upgradeable Contract: This vulnerability occurs when a contract relies on external contracts for critical functions and the external contracts can be dynamically updated [6]. This vulnerability cannot be avoided in Vyper either. The developers have to ensure that they do not outsource critical functions to untrusted external contracts which are built such that their functionality can be dynamically updated. In the future, tools and utilities may be developed that traverse the entire call hierarchy of the contract and its callees to identify functionality which is susceptible to being updated but we are not aware of any such software available at the time of writing.

2) Type casts: This vulnerability occurs due to the Solidity compiler flagging some type errors (e.g., assigning an integer value to a string type) but not all [6]. Types are also used in direct calls, where the caller must declare the callee's interface and cast to it the callee's address when performing the call. Having some type checks may mislead the programmer to believe that all type checks are made. If the function being called doesn't exist in the callee contract then the callee contract's fallback is executed without any exception being thrown to alert the programmer. The functionality is the same in Vyper. Fig. 2 shows an example of a smart contract which defines two contract interfaces (`LibA` and `LibB`) but will have no way of knowing if the argument address passed to function `workerFunction` is of type `LibA` and not `LibB`. Smart contract development tools such as VeriSolid [25] can be used to build and deploy smart contracts that are free from this vulnerability in Solidity.

3) Insufficient signature information: This vulnerability occurs in a contract that uses a signed message to authorize payments to participants (e.g., a micropayment channel contract [44]) and the signed message can be used by the participants to claim authorization for a second action (replay attack). This vulnerability can result in replay attacks in the same contract, across multiple contracts, or even across multiple blockchains. This vulnerability was exploited for

```
1  contract LibA:
2      def featureFunc(arg: uint256): constant
3
4  contract LibB:
5      def featureFunc(arg: uint256): constant
6
7  @public
8  def workerFunction(inputLib: address):
9      LibA(inputLib).featureFunc(1)
```

Fig. 2. Type-cast vulnerability in Vyper.

cross-blockchain replay attacks after the Ethereum classic hard fork [45] and was addressed by the Ethereum Improvement Proposal (EIP) 155 [46]. To avoid this vulnerability within the same contract or across multiple contracts, the developer has to ensure that the contract's signed message generation and authentication mechanism is properly implemented. This can be achieved by including the requisite information (e.g., nonce and contract address) in the message [47]. The developer can also rely on trusted standards like the ERC-721 [48] when implanting token functionality to avoid these problems.

4) Frozen Ether: This vulnerability is observed when Ether is stuck in a contract with no way to send it to other contracts or external accounts. This can happen due to the contract having a faulty or nonexistent function for sending Ether. It can also occur due to the contract relying on another contract for its money-spending functions and the callee contract having been deleted or not being usable anymore. Since this can occur due to a wide range of reasons we believe it is best addressed by the required due diligence on the developer's part and by not outsourcing critical spending functions to untrusted contracts. Third party development and verification tools such as VeriSolid [25] can be used to ensure that the appropriate withdrawal functions always remain reachable in the developed smart contracts.

5) Authentication through `tx.origin`: The `tx.origin` variable is used in Solidity as well as Vyper to refer to the original external account that initiated the transaction in question, whereas the `msg.sender` variable is used in both to refer to the sender of the message for the current call. This vulnerability occurs when an inexperienced developer mistakenly checks `tx.origin` for authentication purposes rather than `msg.sender` [49]. Fig. 3 provides an example of this vulnerability (Line 9) in function `withdrawAll`, which uses `tx.origin` to confirm the owner of the contract that is calling the function. This is an error on the developer's part due to their inadequate understanding of the Ethereum system and the Solidity / Vyper language.

6) Unprotected suicide: This vulnerability occurs due to the fact that contract bytecode and storage can be deleted from the Ethereum network by using the `SELFDESTRUCT` opcode. Both Solidity and Vyper provide functions to use this bytecode. Many contracts implement a self-destruct/suicide function. Developers have to ensure that the authentication mechanism is correctly implemented in such contracts so that only the owner and trusted third parties are able to self-

```
 1  owner: address
 2
 3  @public
 4  def __init__():
 5      self.owner = msg.sender
 6
 7  @public
 8  def withdrawAll(to: address):
 9      assert tx.origin == self.owner
10      send(to, self.balance)
```

```
 1  contract UserWallet:
 2      def withdrawAll(recipient: address):
          ↪ modifying
 3
 4  attacker: address
 5
 6  @public
 7  def payFunc():
 8      UserWallet(msg.sender).
          ↪ withdrawAll(self.attacker)
```

Fig. 3. `tx.origin` misuse in Vyper.

destruct the contract. Developers must also ensure that their contracts do not depend on third party contracts that might be deleted in the future rendering their own contracts unusable. Development and verification tools such as VeriSolid [25] ensure that the suicide statement in smart contracts cannot be reached using an unintended execution trace.

7) Leaking Ether to arbitrary address: This vulnerability exists when a contract is able to send funds to a caller who is not an owner or investor or a legitimate payee of the contract. It occurs due to the contract not enforcing adequate authorization mechanisms before transferring funds or can occur as a result of the many other vulnerabilities mentioned in this survey. This vulnerability can be mitigated by the developer adapting proper authorization logic in code to ensure that only the intended recipients are able to withdraw Ether because the language is blind to the intentions of the developer.

8) Secrecy failure: This vulnerability occurs when developers incorrectly assume that restricting a variable / function's visibility would make its value/functionality hidden from the participants of the Ethereum network [50]. This is not the case due to the public nature of the blockchain. If a state variable is declared private, other contracts are not allowed to access it but participants can still see its value from transaction data. Similarly, the inner-workings of a private function are also visible to all. Hence, the vulnerability is only mitigated if the developer has an understanding of the underlying Ethereum system and cannot be addressed by the language or toolchain.

9) Outdated compiler version: This vulnerability occurs when a contract is compiled using an outdated compiler version which might contain unresolved bugs and vulnerabilities. This vulnerability is addressed by using the latest compiler version when compiling contracts in either Solidity or Vyper.

IV. CONCLUSION

We presented a detailed comparison of how the known vulnerabilities that exist in the Solidity smart contract de-

velopment environment translate to the Vyper development environment. We believe that most of the vulnerabilities listed in Chen et al. [6] are either not addressable at the language / toolchain level or have already been mitigated in Solidity through its subsequent releases and do not surface in Vyper's environment. Vyper may introduce additional vulnerabilities of its own but those will only become evident once a stable version of Vyper is released and adapted by a larger development community. Based on this survey it now appears that most of the vulnerabilities that can be attributed to the language / toolchain have been addressed in Vyper, albeit at the cost of complex functionality. Most remaining vulnerabilities are either due to developers not following the recommended development practices and safety precautions or due to them having insufficient knowledge of the underlying Ethereum system.

REFERENCES

[1] V. Buterin, "Ethereum: a next generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, 2013, accessed on February 26th, 2020.

[2] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[3] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, 2008.

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project – Yellow Paper, Tech. Rep. EIP-150, April 2014.

[5] Ethereum, "Solidity documentation," https://solidity.readthedocs.io/en/latest/, accessed on February 26th, 2020.

[6] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on Ethereum systems security: Vulnerabilities, attacks and defenses," *arXiv preprint arXiv:1908.04507*, 2019.

[7] Ethereum, "Vyper documentation," https://vyper.readthedocs.io/en/v0.1.0-beta.15/, accessed on February 26th, 2020.

[8] King of the Ether, "Post-mortem investigation," https://www.kingoftheether.com/postmortem.html, February 2016.

[9] P. Humiston, "Smart contract attacks [part 2] – ponzi games gone wrong," Hackernoon, https://hackernoon.com/smart-contract-attacks-part-2-ponzi-games-gone-wrong-d5a8b1a98dd8, July 2018.

[10] D. Siegel, "Understanding The DAO attack," CoinDesk, https://www.coindesk.com/understanding-dao-hack-journalists, July 2016.

[11] K. Finley, "A $50 million hack just showed that the DAO was all too human," Wired https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, June 2016.

[12] S. Palladino, "The Parity Wallet hack explained," OpenZeppelin blog, https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, July 2017.

[13] L. H. Newman, "Security news this week: $280m worth of Ethereum is trapped thanks to a dumb bug," WIRED, https://www.wired.com/story/280m-worth-of-ethereum-is-trapped-for-a-pretty-dumb-reason/, November 2017.

[14] Morisander, "The biggest smart contract hacks in history or how to endanger up to US $2.2 billion," https://medium.com/solidified/the-biggest-smart-contract-hacks-in-history-or-how-to-endanger-up-to-us-2-2-billion-d5a72961d15d, March 2018.

[15] National Vulnerability Database, "Cve-2018-10299," https://nvd.nist.gov/vuln/detail/CVE-2018-10299, April 2018.

[16] C. Panda, "The $3 million winner of Fomo3D is still playing to win," LONGHASH, https://en.longhash.com/news/the-3-million-winner-of-fomo3d-is-still-playing-to-win, August 2018.

[17] D. Palmer, "SpankChain loses $40k in hack due to smart contract bug," CoinDesk, https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug, Octorber 2018.

[18] P. Merriam, "Update on the Vyper compiler," Ethereum Blog, https://blog.ethereum.org/2020/01/08/update-on-the-vyper-compiler/, January 2020.

[19] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2016, pp. 254–269.

[20] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[21] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[22] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 653–663.

[23] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC)*, February 2018.

[24] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security (FC)*, February 2019.

[25] K. Nelaturu, A. Mavridou, A. Veneris, and A. Laszka, "Verified development and deployment of multiple interacting smart contracts with VeriSolid," in *Proceedings of the 2nd IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, May 2020.

[26] M. Di Angelo and G. Salzer, "A survey of tools for analyzing Ethereum smart contracts," in *Proceedings of the 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019.

[27] D. Kirillov, O. Iakushkin, V. Korkhov, and V. Petrunin, "Evaluation of tools for analyzing smart contracts in distributed ledger technologies," in *Proceedings of the 19th International Conference on Computational Science and Its Applications (ICCSA)*. Springer, 2019, pp. 522–536.

[28] J. Liu and Z. Liu, "A survey on security verification of blockchain smart contracts," *IEEE Access*, vol. 7, 2019.

[29] D. Harz and W. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," *arXiv preprint arXiv:1809.09805*, 2018.

[30] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON)*, 2018.

[31] OpenZeppelin, "OpenZeppelin Contracts: Library for secure smart contract development – SafeMath," GitHub, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol, 2020.

[32] J. Feist, "Watch your language: Our first Vyper audit," Security Boulevard, https://securityboulevard.com/2019/10/watch-your-language-our-first-vyper-audit/, October 2019.

[33] Vyper Documentation, "Low level built in functions," https://vyper.readthedocs.io/en/latest/built-in-functions.html#low-level-built-in-functions, accessed on February 26th, 2020.

[34] Vyper, "Vyper GitHub repository," https://github.com/vyperlang/vyper/blob/e244d842e883666540f126b7fdfd8a3177341ba4/vyper/functions/functions.py#L740, January 2020.

[35] Vyper Documentation, "Non-reentrant functions," https://vyper.readthedocs.io/en/latest/structure-of-a-contract.html?highlight=reentrance#non-reentrant-functions, accessed on February 26th, 2020.

[36] OpenZeppelin, "OpenZeppelin Contracts: Library for secure smart contract development – Reentrancy Guard," GitHub, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol, 2020.

[37] Solidity Security Blog, "Solidity security: Unexpected ether," GitHub, https://github.com/sigp/solidity-security-blog#3-unexpected-ether-1, 2018, accessed on February 26th, 2020.

[38] ConsenSys Diligence, "Known attacks – DoS with unexpected revert," Ethereum Smart Contract Best Practices, https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-unexpected-revert, accessed on February 26th, 2020.

[39] Ethereum community, "Safety – Ethereum contract security techniques and tips – Favor pull over push for external calls," https://github.com/ethereum/wiki/wiki/Safety#favor-pull-over-push-for-external-calls, accessed on February 26th, 2020.

[40] Solidity Documentation, "Solidity v0.5.0 breaking changes," https://solidity.readthedocs.io/en/v0.5.0/050-breaking-changes.html, accessed on February 26th, 2020.

[41] Solidity Security Blog, "Solidity security: Uninitialised storage pointers," GitHub, https://github.com/sigp/solidity-security-blog#storage, 2018, accessed on February 26th, 2020.

[42] SmartContractSecurity, "SWC-118: Incorrect constructor name," SWC Registry, https://swcregistry.io/docs/SWC-118, accessed on February 26th, 2020.

[43] Solidity Documentation, "Solidity version 0.4.22 release notes," https://github.com/ethereum/solidity/releases/tag/v0.4.22, accessed on February 26th, 2020.

[44] ——, "Solidity by example – Micropayment channel," https://solidity.readthedocs.io/en/v0.5.3/solidity-by-example.html#micropayment-channel, accessed on February 26th, 2020.

[45] baizhenxuan, "Replay attacks on Ethereum smart contracts," GitHub, https://github.com/nkbai/defcon26/blob/master/docs/Replay\%20Attacks\%20on\%20Ethereum\%20Smart\%20Contracts.md, September 2018, accessed on February 26th, 2020.

[46] V. Buterin, "Simple replay attack protection," Ethereum Project, Tech. Rep. EIP-155, October 2016. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md

[47] S. Marx, "Signing and verifying messages in Ethereum," Program the Blockchain, https://programtheblockchain.com/posts/2018/02/17/signing-and-verifying-messages-in-ethereum/, February 2018.

[48] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "ERC-721: Non-fungible token standard," Ethereum Improvement Proposals, Tech. Rep. EIP-721, January 2018.

[49] P. Vessenes, "Tx.Origin and Ethereum oh my!" https://vessenes.com/tx-origin-and-ethereum-oh-my/, June 2016, accessed on February 26th, 2020.

[50] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*. Springer, April 2017, pp. 164–186.