CHALMERS
UNIVERSITY OF TECHNOLOGY

Short Paper: Blockcheck the Typechain

Citation for the original published paper (version of record):
Benitez, S., Cogan, J., Russo, A. (2020). Short Paper: Blockcheck the Typechain. PLAS 2020 - Proceedings of the 15th Workshop on Programming Languages and Analysis for Security, 13 November 2020: 35-39. http://dx.doi.org/10.1145/3411506.3417600

Short Paper: Blockcheck the Typechain

Sergio Benitez
sbenitez@stanford.edu
Stanford University
Stanford, CA, USA

Jonathan Cogan
jcogan2@stanford.edu
Stanford University
Stanford, CA, USA

Alejandro Russo
russo@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Abstract

Recent efforts have sought to design new smart contract programming languages that make writing blockchain programs safer. But programs on the blockchain are beholden only to the safety properties enforced by the blockchain itself: even the strictest language-only properties can be rendered moot on a language-oblivious blockchain due to inter-contract interactions. Consequently, while safer languages are a necessity, fully realizing their benefits necessitates a language-aware redesign of the blockchain itself.

To this end, we propose that the blockchain be viewed as a *typechain*: a chain of typed programs — not arbitrary blocks — that are included *iff* they typecheck against the existing chain. Reaching consensus, or *blockchecking*, validates typechecking in a byzantine fault-tolerant manner. Safety properties traditionally enforced by a runtime are instead enforced by a type system with the aim of statically capturing smart contract correctness.

To provide a robust level of safety, we contend that a typechain must minimally guarantee (1) asset linearity and liveness, (2) physical resource availability, including CPU and memory, (3) exceptionless execution, or no early termination, (4) protocol conformance, or adherence to some state machine, and (5) inter-contract safety, including reentrancy safety. Despite their exacting nature, typechains are extensible, allowing for rich libraries that extend the set of verified properties. We expand on typechain properties and present examples of real-world bugs they prevent.

CCS Concepts

• **Software and its engineering → Language features**; Syntax; *Semantics*; *Compilers*; *State based definitions*; *Domain specific languages*; • **Security and privacy → Software security engineering**; **Software and application security**.

Keywords

blockchain, security, safety, typechain, programming language, type system, compiler, smart contract, digital currency

ACM Reference Format:
Sergio Benitez, Jonathan Cogan, and Alejandro Russo. 2020. Short Paper: Blockcheck the Typechain. In *15th Workshop on Programming Languages and Analysis for Security (PLAS'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3411506.3417600

PLAS'20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8092-8/20/11.
https://doi.org/10.1145/3411506.3417600

1 Introduction

Smart contracts are rife with bugs. Invariably, these bugs result in the loss of dollars, at times measured in the hundreds of millions. Many of these bugs can be traced back to design decisions, inconsistencies, and footguns in the contract's programming language [3, 13, 26, 31, 45]. Evidently, we should improve our smart contract programming facilities.

Recent efforts in industry and academia have sought to design new, strongly typed smart contract languages to assuage these concerns. While a step forward, we maintain that these efforts include or lack key language properties that will inevitably lead to similar bugs. For example, Flint [34], Scilla [35], and Obsidian [8] abort execution due to runtime faults, a design decision that can lead to contracts that inadvertently lock funds indefinitely [11, 30, 39]. Neither Bamboo [44] nor Vyper [40] enforce asset linearity, allowing for asset loss or duplication as well as inconsistency between the accounting and presence of value, two common pitfalls [3, 11, 12]. And while Nomos [10] excels at avoiding these prior deficiencies, it limits inter-contract interactions and is susceptible to deadlocks when they do occur.

Consequently, we hold that smart contract programming languages necessitate a type system with properties that specifically capture smart contract correctness. In particular, typechecking in such a language should minimally verify:

(1) **asset linearity and liveness**, to prevent the loss, duplication, and inaccessibility of value
(2) **physical resource availability**, including CPU and memory, to ensure the possibility of executing to completion
(3) **exceptionless execution**, or lack of unchecked early termination, to enforce complete error handling
(4) **protocol conformance**, or adherence to some state machine, to prevent executing code in unintended states
(5) **inter-contract safety**, to prevent reentrancy bugs [12] while allowing safe inter-contract interactions

But a language-only approach is insufficient: failure to consider inter-contract interactions precludes verification of these desired properties. For example, to verify a contract exception-free, *every* contract it interacts with must be exception-free. Similarly, for *any* contract to soundly treat an asset linearly, *all* contracts must treat that asset linearly. Thus, realizing strict language properties on the blockchain requires the blockchain itself to be language-aware.

To this end, we propose a type-driven, language-aware redesign of the blockchain [29, 42] as a *typechain*: a chain of typed contracts, state, and transactions — not arbitrary blocks — that are accepted *iff* they typecheck against the existing chain. The typechain is *blockchecked* by miners to validate typechecking, verifying, at minimum, the aforementioned properties.

While typechains have stronger properties than many strongly typed languages, they may not sufficiently capture all desired smart-contract correctness properties. Thus, a typechain must be extensible, allowing for rich libraries that extend its set of verified properties. For example, whereas prior work has developed entirely new blockchains to enable on-chain privacy [5, 7, 9, 16, 20] or verified off-chain computation [2, 4, 18, 22, 24, 27], a typechain allows this functionality to be implemented as a library without modification to its core protocols or language.

In the following section, we expand on typechains, detailing their desired properties, and present examples of real-word bugs they preclude (§2). Following, we illustrate how libraries can extend a typechain's set of verified properties (§3). Finally, we provide concluding remarks (§4).

2 Formulating the Typechain

A *typechain* is an extensible sequence of statically typed programs ordered and grouped into blocks. To add a block to the chain, it must be *blockchecked*: a distributed network of machines must agree that the program formed by adjoining the new block is well-typed.

Formally speaking, a program $p = \overline{d; s}$ on the typechain is a sequence of typed declarations d and closed statements s. The typechain is a sequence of programs[1] $C_\tau = \overline{p}$ where

$$\forall p \in C_\tau.\texttt{blockcheck}(p) \iff \texttt{blockcheck}(C_\tau).$$

Newly chained programs — contracts or transactions — must typecheck individually, and the program formed by the concatenation of existing programs and the new program must also typecheck. Thus, successfully blockchecking a new program p can be seen as swapping some existing typechain C_τ for a new typechain C'_τ where

$$C'_\tau = C_\tau; p \quad \textit{if} \quad \texttt{blockcheck}(C'_\tau).$$

From a different yet isomorphic perspective, a typechain is a refinement (by one account) or restriction (by another) of a blockchain in which miners' core task is to reach consensus on the type safety of the program resulting from the concatenation of all programs on the chain. All safety properties, which traditionally arise from the blockchain's runtime [42], arise instead from the type system. This includes asset accounting, proof and ownership of assets, and the ability, financial or otherwise, to compute or store on the blockchain. The deployment of a contract or execution of a transaction is preconditioned on these properties.

A typechain's type system is thus stronger than that of many conventional strongly typed languages. Minimally, it must guarantee the properties enumerated in §1. We emphasize that, while helpful, a subset of these properties is *not* sufficient; a single omission is enough to admit the very class of nefarious, money-squandering bugs we wish to avoid. Conversely, this minimal set of properties is necessarily insufficient to capture all desirable correctness properties, so typechains are extensible, allowing for new, arbitrary properties to be admitted and enforced (§3).

In the following sections, we detail the properties in §1, providing examples of bugs that each would prevent. Our examples are written in pseudocode resembling Rust [33, 36].

[1] Note that $C_\tau = \overline{p} = \overline{d; s} = p$, so the typechain is itself a program.

```
impl Auction when open {
  fn bid(self, amount: Money) {
    if self.should_close() {
      close self;
    } else if amount <= self.max_bid {
      amount.return_to_sender();
    } else {
      self.max_bid = amount;
    }
  }
}
```

Figure 1: A buggy `Auction` contract with a `bid()` method.

Asset Linearity and Liveness

A blockchain is primarily concerned with managing and operating on assets: currency, arbitrary items, tokens (sub-currencies), and so on. As in the real world, correctness demands that assets are never lost nor duplicated.

Unfortunately, this property is difficult to maintain in existing blockchain languages [3, 12, 13], where assets are trivially reusable and dispensable. As an example of what can go wrong, consider the `bid()` method in Figure 1. The intent is that, while the auction is open, bids of `amount` can be placed by invoking `Auction::bid(amount)`. The sender becomes the highest bidder if their bid is the highest thus far. Otherwise, the bid is returned. Can you spot the problem(s)?

The first branch, which closes the auction when necessary, fails to return the unconsidered bid `amount` to its bidder. The third branch, which runs if `amount` is the highest bid, overwrites `max_bid` without returning this previous highest bid to *its* bidder. In both cases, value is irreversibly lost.

A fix is to call the appropriate `return_to_sender()` in each branch. A language with linear assets would statically enforce the existence of such calls. Linear types [17, 41] provide a mechanism: encode assets as linear types, enforcing their use *exactly once*, preventing loss or duplication.

However, linearity alone is insufficient: typechains must also enjoy *asset liveness*, or the guarantee that contracts are always in a state that allows their assets to be transferred. Failure to enjoy asset liveness can result in assets that are *locked* and rendered unusable. One famous instance of such a bug led to the loss of ~200 million USD [32], though many other instances have occurred in the wild [3, 11, 13].

Physical Resource Availability

To protect miners from executing runaway transactions, themselves arbitrary programs, blockchains enforce limits on computational resources including the CPU and memory. Overstepping transactions are halted or partially reverted.

This behavior can lead to unexpected consequences, especially when state is only partially reverted [11, 23, 39]. As an example, we consider again `bid()` in Figure 1. In languages like Solidity [15], *send* calls like `return_to_sender()` can fail if such a call would exhaust the stack limit or if other alloted computational resources would not suffice. If the call fails, a `false` value is returned [3, 15]. Unfortunately, contracts often ignore this value, leading to a class

of bugs known as *unchecked send* [3, 45]. An unchecked send in the `bid()` method, for example, would result in the `Auction` contract inadvertently holding more funds than desired.

Contracts may also settle in a state where execution without exhausting resources, and thus execution at all, is impossible [30]. Such a contract has its funds locked forever. This entails that *asset liveness* necessitates *resource availability*. As an example, GovernMental's only asset extraction method iterated over an array that grew too large to scan within the alloted limit; its funds were locked without recourse [39].

A typechain statically guarantees the availability of physical resources for all possible executions of a contract. Mechanically, its type system produces concrete, input-independent resource usage upper bounds. Programs exceeding limits fail to blockcheck. While languages like Nomos [10] and Scilla [35] demonstrate feasibility of such a guarantee, a challenge remains in finding mechanisms that are developer-friendly.

Exceptionless Execution

In addition to resource exhaustion, early termination in existing blockchains can arise due to arbitrary unchecked exceptions [25, 38]. In Solidity, contracts can arbitrarily `throw` exceptions, and in Solidity, Flint, and Obsidian, exceptions can arise from incorrect logical operations such as out-of-bounds indexing and division by zero. While their sources of early termination differ, unchecked exceptions and resource exhaustion admit similar issues. Thus, a typechain must either guarantee that all possible exceptions are checked or prohibit exceptional operations entirely, perhaps by leveraging dependent types [43] or SMT solvers [1].

Protocol Conformance

Smart contracts typically follow a *protocol*: a predefined series of steps conditioned on the contract's state. For instance, `Auction` in Figure 1 allows a `bid` only when it is `open`.

Existing blockchains require programmers to enforce protocols in an ad-hoc, unstructured, and dynamic manner [14]. Often, they advocate for throwing exceptions when expected conditions fail to hold, violating *exceptionless execution*. Of course, mistakes abound in practice [13, 25, 38]. Typechains, instead, allow programmers to directly encode a contract's protocol; conformance to the protocol is then statically enforced by the type system.

Recent efforts have recognized the importance of protocol conformance: Obsidian [8], Flint [34], and Nomos [10] all statically enforce protocols, the former two by leveraging typestate [37] and the latter by leveraging session types [19].

Inter-Contract Safety

Blockchains like Ethereum allow smart contracts to interact through mutual method invocation. While largely innocuous, interaction can lead to vulnerabilities in both the caller and callee. For instance, given a stack frame limit of k frames[2], a caller can coerce a callee into exhausting its frame limit by making $k-2$ nested calls prior to invoking the target callee. When the callee makes a call of its own, it will exceed the frame limit and an exception will be thrown. As discussed, if unexpected, such exceptions can lead to vulnerabilities.

[2] In Ethereum, $k = 1024$.

Callers are inherently vulnerable. Since a callee can react arbitrarily, it may re-invoke the caller. If the caller expected logically atomic execution, its re-execution may result in inconsistent state which in turn may lead to vulnerabilities. Such vulnerabilities are termed *reentrancy* vulnerabilities.

The issue is further exacerbated by the ability for contracts to interact unknowingly. In Ethereum, for instance, asset-transfer calls, such as `return_to_sender()` in Figure 1, invoke a method on the recipient's contract. Infamously, ~40 million USD was stolen from TheDAO via a reentrancy attack resulting from such a call [12]. The attack was considered so unacceptable that Ethereum forked, mutating the purportedly immutable blockchain in order to make the stolen funds recoverable [28]. The particularly vile nature of reentrancy vulnerabilities has led to the development of myriad tools and methods to detect them [6, 21, 25, 30, 38]. These approaches require manual application by the programmer, and to date, none is both sound and complete.

A typechain enables safe inter-contract communication by guaranteeing that *local* contract reasoning extends *globally*. This is facilitated by the properties enumerated thus far. For example, exceptionless execution and resource availability immediately obviate caller-originated resource exhaustion attacks, while asset linearity ensures that inter-contract interactions, even reentrant ones, cannot render asset accounting logically inconsistent, the crux of TheDAO attack. In addition, a typechain's type system should be statically aware of all inter-contract interactions and conservatively forbid reentrant calls, perhaps by leveraging session types [19] or communicating automata [35].

3 Extending the Typechain

A typechain's properties are *necessary* but *insufficient*: a smart contract is correct *only if* these properties hold, but desired correctness properties may be absent.

To compensate, typechains are malleable: a typechain's type system is extensible (§3.1) and allows for rich libraries that leverage Curry-Howard isomorphisms to expose types that act as proof witnesses (§3.2). Thus, a typechain's type system must at once be strict enough to allow enforcing arbitrary correctness properties while permissive enough to host expressive libraries.

3.1 Type System Strengthening

Smart contracts can extend the set of properties verified by a typechain via extensions that *strengthen* the typechain's type system. Extensions run during blockchecking and apply to the contract that deployed the extension as well as its dependents. Properties checked by extensions are guaranteed to hold uniformly across the entire typechain. Any property that can be expressed as a finite static analysis can be implemented as an extension.

Concretely, extensions are type systems that are required to be *strengthenings* of the typechain's type system. We say that a type system defined by the relation $\Gamma' \vdash' e : \tau$ is a *strengthening* of a type system defined by $\Gamma \vdash e : \tau$ *iff* $\Gamma' \vdash' e : \tau \implies \Gamma \vdash e : \tau$.

We call $\Gamma' \vdash' e : \tau$ the *strengthened* type system; $\Gamma \vdash e : \tau$ is the *original* type system. A strengthened type system admits monotonically fewer programs than the original. In other words, it reduces

the set of programs that typecheck. Critically, a strengthening cannot weaken a type system: a property that holds in the original must hold after strengthening. Consequently, extensions cannot deliberately or accidentally weaken a typechain's properties.

Example: Global Singleton

As an example of a property verifiable via extensions, consider the *global singleton*: a type with at most one instance in a program. We would like to implement a type, `Singleton`, external to the typechain, that is guaranteed to be a global singleton in the typechain. In other words, we would like to verify that at most one instance of type `Singleton` is ever constructed in the typechain.

To begin, the `Singleton` type is declared as the usual container generic over an embedded type `T`:

```
struct Singleton<T>(T);
```

We define a flow-sensitive strengthening of $\Gamma \vdash e : \tau$ as

$$\frac{\Gamma \vdash e : \tau \quad \neg \texttt{singleton}(\tau)}{\Gamma, \Delta \vdash_\Delta e : \tau \Rightarrow \Delta}\,0 \qquad \frac{\Gamma, \Delta \vdash_\Delta v : \tau \Rightarrow \Delta' \quad \tau \notin \Delta'}{\Gamma, \Delta' \vdash_\Delta v : \tau \Rightarrow \Delta' \cup \{\tau\}}\,1,$$

where `singleton`(τ) is `true` when τ = `Singleton<C>` for some concrete type `C`, and v is a value. Rule 0, to the left, ensures that all non-`Singleton` types typecheck as usual. Rule 1, to the right, asserts that a `Singleton` instance can only arise from one value in the program. Assuming instances must be constructed as values v, this extension guarantees the *global singleton* property we sought.

3.2 Type Libraries

While many existing type systems are readily leveraged to codify correctness properties, the practice is notably absent in blockchain programming, owing to weaker type systems. Typechains nullify this dichotomy by being flexible yet strict enough to allow such properties to be codified in libraries.

Example: Timing

We consider *time-based* properties as an example, where correctness is specified as a function of time. A typechain should allow the implementation of a library contract type `Timed<C>`, generic over a contract `C`, that automatically and indelibly enforces centrally declared, time-based properties. Leveraging such a library for `Auction` in Figure 1 might resemble:

```
impl Timing for Auction {
  fn check(self, time: Time) {
    if self.should_close(time) close self;
  }
}
```

This `Timing` implementation specifies time-based properties for `Auction`: if an auction `should_close()`, it is, in fact, closed. By deploying a `Timed<Auction>` in place of an `Auction`, the library would automatically enforce this property. Consequently, `Auction` can be written without further calls to `self.should_close()`. In particular, methods implemented on `open` contracts, such as `Auction::bid()`, can rely on being called only when the auction is logically open without further confirmation.

4 Concluding Remarks

A *typechain* is a sequence of typed programs *blockchecked* by a distributed network of machines that verify their well-typedness. When the type system ensures key safety properties, bugs plaguing smart contracts cease to be. These properties must minimally include (1) asset linearity and liveness, (2) physical resource availability, (3) exceptionless execution, (4) protocol conformance, and (5) inter-contract safety. Existing work does not go *far enough*, enjoying only a subset of these properties. Unfortunately, a single omission permits the very bugs we attempt to evade.

Verification of these properties must holistically consider all programs on the chain: a single errant contract thwarts soundness. Thus, unlike verification applied to individual contracts, typechains enjoin safety properties universally, requiring them to hold of *all* contracts. Still, a typechain may not sufficiently capture all desired correctness properties, so typechains are extensible, allowing user-defined properties to be verified during blockchecking. By blockchecking safety and user-defined correctness properties, the ambit of verification is approached.

We believe typechains stand to fundamentally improve the safety and correctness of smart contracts and are actively working on the design and implementation of a typechain.

Acknowledgments

Thanks to Dan Boneh, Fraser Brown, and the anonymous reviewers for their invaluable feedback and suggestions.

References

[1] Leonardo Alt and Christian Reitwiessner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 376–388.

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. 443–458.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag, Berlin, Heidelberg, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[4] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2018. Zexe: Enabling Decentralized Private Computation. *IACR Cryptol. ePrint Arch.* 2018 (2018), 962. https://eprint.iacr.org/2018/962

[5] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. 2018. Private Data Objects: an Overview. *CoRR* abs/1807.05686 (2018). arXiv:1807.05686 http://arxiv.org/abs/1807.05686

[6] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981 [cs.PL]

[7] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (Jun 2019). https://doi.org/10.1109/eurosp.2019.00023

[8] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. arXiv:1909.03523 [cs.PL]

[9] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86. http://eprint.iacr.org/2016/086

[10] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2019. Resource-Aware Session Types for Digital Contracts. arXiv:1902.06056 [cs.PL]

[11] David Gerard. 2017. *Attack of the 50 Foot Blockchain.* David Gerard, Chapter 10.

[12] David Siegel. 2016. *Understanding The DAO Attack* . https://www.coindesk.com/understanding-dao-hack-journalists

[13] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2015. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptol. ePrint Arch.* 2015 (2015), 460. http://eprint.iacr.org/2015/460

[14] Ethereum Project Developers. 2020. *Solidity - Common Patterns - State Machine.* https://solidity.readthedocs.io/en/v0.6.10/common-patterns.html#state-machine

[15] Ethereum Project Developers. 2020. *Solidity - Solidity Documentation.* https://solidity.readthedocs.io/

[16] Hisham S. Galal and Amr M. Youssef. 2019. Trustee: Full Privacy Preserving Vickrey Auction on top of Ethereum. *CoRR* abs/1905.06280 (2019). arXiv:1905.06280 http://arxiv.org/abs/1905.06280

[17] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[18] S Goldwasser, S Micali, and C Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178

[19] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science)*, Chris Hankin (Ed.), Vol. 1381. Springer, 122–138. https://doi.org/10.1007/BFb0053567

[20] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1353–1370. https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[22] Gabriel Kaptchuk, Matthew Green, and Ian Miers. 2019. Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/giving-state-to-the-stateless-augmenting-trustworthy-computation-with-ledgers/

[23] King of the Ether Throne Developers. 2016. *King of the Ether Throne Post-Mortem Investigation.* http://www.kingoftheether.com/postmortem.html

[24] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2015. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. *IACR Cryptol. ePrint Arch.* 2015 (2015), 675. http://eprint.iacr.org/2015/675

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[26] Martin Holst Swende. 2015. *An Ethereum Roulette.* https://swende.se/blog/Breaking_the_house.html

[27] Izaak Meckler and Evan Shapiro. 2018. Coda: Decentralized cryptocurrency at scale. (2018).

[28] Michael del Castillo. 2016. *Ethereum Executes Blockchain Hard Fork to Return DAO Funds.* https://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds

[29] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report.

[30] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[31] Parity Technologies. 2017. *A Postmortem on the Parity Multi-Sig Library Self-Destruct.* https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/

[32] Parity Technologies. 2017. *The Multi-sig Hack: A Postmortem.* https://www.parity.io/the-multi-sig-hack-a-postmortem/

[33] Rust Project Developers. 2020. *Rust Programming Language.* https://www.rust-lang.org

[34] Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. 2019. Flint for Safer Smart Contracts. *CoRR* abs/1904.06534 (2019). arXiv:1904.06534 http://arxiv.org/abs/1904.06534

[35] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360611

[36] Steve Klabnik and Carol Nichols, with contributions from the Rust Community. 2020. *The Rust Programming Language.* https://doc.rust-lang.org/book/#the-rust-programming-language

[37] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[38] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[39] u/ethererik. 2016. *GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas.* https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/

[40] Vitalik Buterin. 2020. *Vyper: a contract-oriented, pythonic programming language that targets the Ethereum Virtual Machine (EVM).* https://vyper.readthedocs.io/en/latest/

[41] Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy (Ed.). North-Holland, 561.

[42] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[43] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. https://doi.org/10.1145/292540.292560

[44] Yoichi Hirai. 2020. *Bamboo: a language for morphing smart contracts.* https://github.com/CornellBlockchain/bamboo

[45] Zikai Alex Wen and Andrew Miller. 2016. *Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug.* https://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/