Not All Code are Create2 Equal

Michael Fröwis and Rainer Böhme

Department of Computer Science, Universität Innsbruck, Austria

Abstract. We describe the impact and measure the adoption of the `CREATE2` instruction introduced to the Ethereum Virtual Machine in the Constantinople upgrade. This change to Ethereum's execution environment is fundamental because it enables to modify the program stored on a given address after deployment, making it much harder to reason about the immutability of smart contracts. We enumerate six use cases and novel attack vectors, and present empirical evidence from all 32 million code accounts created between March 2019 and July 2021. The data shows that the main beneficiaries of the upgrade are wallet contracts, which can now use predictable addresses. But they do not require the more risky feature of mutable smart contracts. So far, the only applications that use the latter are front-running bots and gas tokens.

Keywords: Smart Contracts, Patching, Security, Ethereum Virtual Machine

1 Introduction

The initial design of the Ethereum Virtual Machine (EVM) stipulated that program code associated with the address of a code account cannot be changed. This enabled users to build confidence in the integrity of a smart contract before entrusting it any value. A single exception, the possibility to remove code with the instruction `SELFDESTRUCT`, has caught public attention in 2017, when it caused a $152 million loss in an incident of a popular wallet [9]. Nonetheless, code removed this way could not be restored or replaced with other code.

The Constantinople upgrade, effective since February 2019,[1] breaks with this convention. It adds `CREATE2`, as defined in EIP-1014, to the EVM's instruction set. The new instruction derives the addresses of code accounts in a more controllable way than the legacy `CREATE` instruction, which is still supported for backward compatibility. Effectively, `CREATE2` enables users to deploy different code to the same address and therefore provides a means to update smart contracts after deployment.

This radical—to some observers unanticipated—change of fundamental functionality has several far-reaching implications. It was motivated by the demand for counterfactual instantiation, i. e., the possibility to commit to program code before it is deployed on the chain. This is useful for the efficient establishment of state channels, a nascent off-chain technology [14,11]. Moreover, the upgrade

[1] 2019-02-28 20:52:04 GMT+2

simplifies the maintenance of smart contracts. Previously, rather expensive proxy patterns were necessary to enable patching [20,15]. These become obsolete if code can be updated right away using `CREATE2`.

On the downside, the Constantinople upgrade made it harder to reason about the code that is governing a given address at any future point in time. Many subtleties not only render it virtually impossible for end users to convince themselves about the immutability of any but the most trivial smart contracts. Also developers are prone to run into pitfalls, and automated tools for smart contract verification may require adaptations [30,1]. The mere fact that not all users may be aware of the possibility to update code raises security concerns.

Ethereum is not the only platform affected. The EVM has become the de facto standard runtime environment for smart contract environments including Ethereum Classic,[2] Binance Smart Chain, xDai, POA, and Polygon. At the time of writing, the value administered on these platforms exceeds $100 billion.

The objective of this paper is twofold. First, it offers a comprehensive description of the new possibilities enabled by `CREATE2` and through the interactions between old and new code. This leads us to identify a set of features comprising new threats as well as new opportunities for applications. Second, the paper takes a measurement approach to empirically review the prevalence of these features on the Ethereum blockchain since the Constantinople upgrade in 2019 until shortly before the London upgrade in August 2021. This measurement is based on novel heuristics, which add to the contributions made in this work.

The rest of the paper is organized as follows. Section 2 describes how smart contracts could be patched before and after the upgrade. Section 3 discusses the effects of the upgrade and motivates the measurement of six items of interest (IoI). Section 4 presents the data sources, methods and proposed heuristics. Section 5 reports and interprets the measurement results along the IoIs. Section 6 connects to relevant related work. Section 7 concludes.

2 Updating Smart Contracts

This section reviews how smart contracts on Ethereum could be updated before and after the Constantinople upgrade.[3] To do so, we introduce some terminology on how code accounts are referenced on the platform. Before the upgrade, addresses of code accounts were bound to the deployer, i.e., the address which has sent the `CREATE` transaction and a deployer specific nonce. We shall call this approach *legacy addresses*. The upgrade introduced an alternative way to reference code accounts, which replaces the nonce with a commitment to code. We call this approach *code-bound addresses*.

[2] Ethereum Classic activated the Constatinople changes in January 2020 [24].

[3] We use *upgrade* to refer to changes of the platform and *update* for the possibility to alter code on the platform.

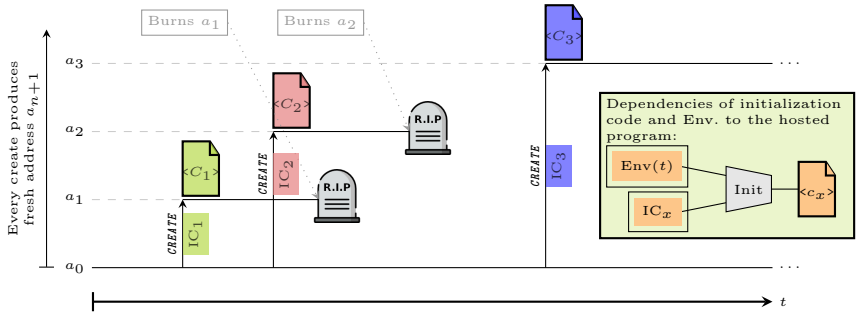


Fig. 1: Pre-Constantinople: addresses are single-use. *Deactivation* "burns" the address, i. e., future reuse is impossible. IC_x stands for initialization code.

2.1 Updating Smart Contracts with Legacy Addresses

For legacy addresses, the address of a new code account is computed as

$$a_{n_d} = \text{createAddr}_{d,n_d}() = (\mathcal{H}(\text{rlp}(d, n_d)))_{0\ldots159}$$

where $\mathcal{H}$ is the 256-bit Keccak hash function, d is the deployer address, n_d is the nonce associated with the deployer address, $\text{rlp}(d, n_d)$ is the Recursive Length Prefix (RLP) encoding of d and n_d, and $(\cdot)_{0\ldots159}$ denotes talking the 160 least significant bits of the value [40]. For externally owned accounts, which are controlled by private keys, n_d is the number of transactions sent. For code accounts, the nonce equals the number of code accounts already created from that address. As nonces increase strictly monotonically, creating the same address twice is unlikely.[4] The Byzantium upgrade[5] introduced a precaution against the residual risk of address collision: account creations fail if either a program or a non-zero nonce is associated with the target address (defined in EIP-684 [10]).

Figure 1 illustrates how new programs are created using legacy addresses. The vertical axis depicts the address space. Every new program is deployed to a new unique address. When a program is *deactivated*, by executing the `SELFDESTRUCT` instruction, the address is burned and cannot be reused. The actual code C hosted at an address a is defined by the output of an *initialization code* IC provided in the `CREATE` transaction. Since the initialization code can access the state of the blockchain, the deployed code can also depend on the environment at the time of creation. In practice, many initialization codes return a constant.

Updating smart contracts created with legacy addresses is complicated. As the code at a given address is fixed, and changing all references to a new address is impractical and risky, developers must prepare their applications for updates by using e. g., the transparent proxy pattern [20,15]. This means splitting a program over multiple code accounts so that the main program is called indirectly from a stub that resides at a permanent address (and thus cannot be updated). The stub

[4] The odds are one in 2^{80} accounts, following from the birthday problem.
[5] Effective since 2017-10-16 07:22:11 GMT+2

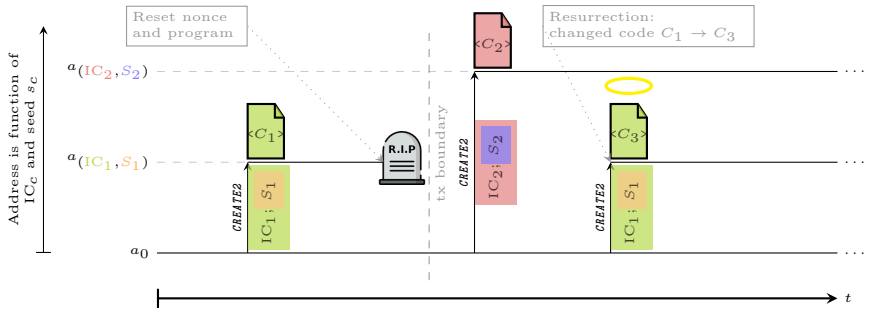


Fig. 2: Post-Constantinople: addresses are reusable. This enables *resurrections*. With the right initialization code, it is possible to change the program code.

stores the address of the current main program in its state, and allows authorized parties to change the value. This way smart contracts become patchable.

Popular libraries, like OpenZeppelin, provide off-the-shelf solutions for this pattern [32]. Several EIPs seek to standardize the proxy pattern [33,21,34,26]. None of the proposals appears to be final and only some of them are used.[6]

2.2 Updating Smart Contracts with Code-bound Addresses

The `CREATE2` instruction differs from `CREATE` in the way it calculates addresses:

$$a(i, s) = \text{create2Addr}_d(i, s) = (\ \mathcal{H}(\ 255 \parallel d \parallel s \parallel \mathcal{H}(i)\ )_{0\ldots159}$$

where s is a 32-byte seed, i is the initialization code, and $\parallel$ denotes concatenation [11]. Code-bound addresses are tied to the initialization code IC and the deployer address, but not on the nonce. The constant 255 prevents collisions with legacy addresses because RLP encodings starting with 255 would imply petabytes of data. The seed s can be chosen freely to enable factory contracts that create multiple code accounts sharing the same initialization code.

Since `CREATE2` does not bind the address to a nonce, creating collisions is easy: use the same deployer account with the same seed and initialization code IC. However, `CREATE2` fails if the target address already hosts a program [10]; unless the code account is *deactivated*. After deactivation, it is possible to recreate a code account on the same address. We call this *resurrection*.[7]

Figure 2 illustrates a resurrection of an address. Observe that since code-bound addresses commit to the initialization code and not the actual program code, it is indeed possible to deploy a different program after a resurrection. For

[6] Using simple heuristics derived from the EIPS we found 223 873 following EIP-897, 0 following EIP-1167, 22 238 following EIP-1822, and 31 432 following EIP-1967.

[7] Deactivation and the resurrection cannot take place in the same transaction because the use of `SELFDESTRUCT` resets the account and its nonce at the very end of the transaction. But it is possible within the same block.

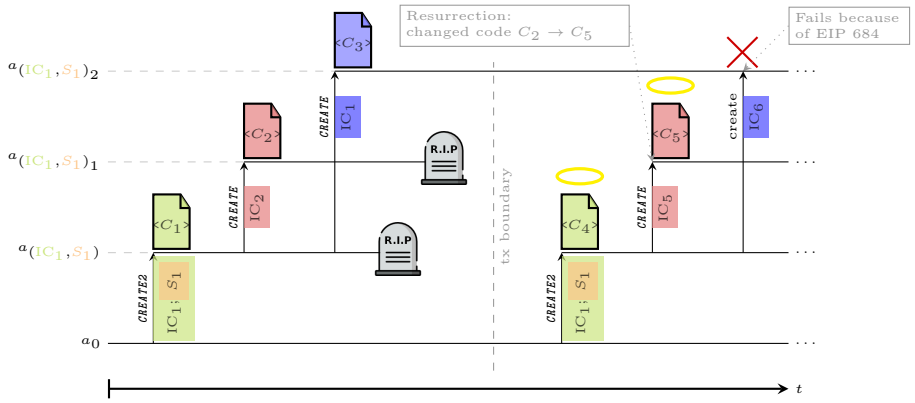


Fig. 3: Post-Constantinople: *cascaded resurrections* of code accounts created with ***CREATE*** are possible because nonces of code accounts are no longer monotonic.

this the output of the initialization code must depend on the blockchain state. We call this behavior *morphing resurrection*.

```
return address(0xABC...DEF).
    get_code();
```

Example 1.1: Morphing resurrection involving another code account

```
if block.number > 1000000 {
  return "0x123...";
} else {
  return "0x345...";
}
```

Example 1.2: Morphing resurrection with simple condition

Example 1.1, sketches initialization code which fetches the program code to be deployed from the code account at address `0xABC...DEF`.[8] Another, less flexible, approach is to include different programs as constants in the initialization code and select the return value based on the state of the environment (e. g., the block height). This is shown in Example 1.2.

2.3 Interaction of *CREATE* and *CREATE2*

Perhaps unexpectedly, the use of ***CREATE2*** makes it possible to resurrect programs created with the ***CREATE*** instruction. This requires some indirections. Recall that ***CREATE2*** can reset the nonce of a code account by resurrecting it. This breaks the monotonicity of nonces which prevented collisions. If the code account we can resurrect with ***CREATE2***, has used ***CREATE*** to generate child code accounts, then these children can be resurrected with ***CREATE*** [35]. Figure 3 sketches this process, which we call *cascaded resurrection*.

The interaction of ***CREATE*** and ***CREATE2*** means that to detect possible resurrections, it is not sufficient to check for the immediate use of ***CREATE2***. Instead, the entire deployer chain must be analyzed. Only code accounts deployed with

[8] We prefer pseudo-code over Solidity because Solidity does not offer a way to encode explicit returns in constructors without the use of inline EVM assembly.

CREATE before the Constantinople upgrade are safe because it was not possible to have a *CREATE2* transactions in the deployer chain.

More generally, while it was possible to detect the proxy pattern (or the absence thereof) by inspecting the deployed code, the Constantinople update made it much more difficult for users to reason about the immutability of smart contacts. Almost two years after the Constantinople upgrade, we are not aware of any block explorer or open blockchain analytics tool offering a convenient way to see the deployment history of a contract, or simply whether it was created using *CREATE* or *CREATE2*. Etherscan does show if a code account had a resurrection. This is a first step, but users need to know in advance if an account is resurrectable.

3 Measurement Motivation

Updating code accounts by resurrection is a fundamental change to the EVM's security model. Attackers can use (cascaded) morphing resurrections to trick users into depositing value into an account hosting seemingly benign code, and replace it with a malicious version before the victim executes a transaction to claim value back. While community sources suggest that this change might have been introduced unintentionally [35,29,38], the question we ask is how the Constantinople upgrade has affected the ecosystem. To do so, we carry out empirical measurements, including—to the best of our knowledge for the first time—a heuristic to detect potentially resurrectable code accounts.

This section decomposes the effects of the upgrade into six items of interest (IoIs), some offering mainly benefits and others increasing the attack surface. The measurement methods and results in later sections will speak to these IoI.

3.1 IoIs Relating to Feature Expansion

Cheaper Patching (i). *CREATE2* makes updateable smart contracts cheaper. Resurrections avoid the overhead of splitting the smart contracts into multiple code accounts, and the communication overhead required for transparent up-gradable proxies [35]. This way, smart contracts can be patched. Users must be wary if patching is desired and if they trust the party who can patch.

Generalized State Channels (ii). Counterfactual instantiation [14], the possibility to commit to settlement code of a state channel before deployment, was the declared intention for adding *CREATE2* to the EVM's instruction set. Code-bound addresses enable exactly this. However, recall from Section 2.2 that the hash is taken over the initialization code that produces the program code. Parties using counterfactual instantiation should check if the counterfactual initialization code is static:

Definition 1. *An initialization code is* ***static*** *if its return value does not depend on the environment at execution time.*

If one party could trick the other in agreeing on non-static initialization code, it could appropriate all funds in the state channel independent of the agreement.

Wallet On-Boarding (iii). Using wallet code accounts instead of externally owned accounts has many advantages. Wallets decouple the authentication from the addresses holding funds. This gives users advanced recovery options in the case of lost keys. To create wallet code accounts, users need funds to pay the transaction fee of the creation. Before the upgrade, this meant users had to have a funded externally owned account to claim a new wallet code account. The upgrade made on-boarding easier by using a counterfactual instantiation of the wallet, which can immediately receive funds.

Vanity Addresses (iv). Some users prefer short addresses, found after grinding through hash pre-images. The main rational is to save gas cost. Vanity addresses can (a) be packed more tightly in case of leading zeros; or (b) are cheaper to pass as call parameters, since zero bytes get a discount [40]. In the absence of a common definition of vanity addresses, we define them as follows:

Definition 2. *A **vanity address** is an address with more than four leading zero bytes.*

Vanity addresses for code accounts existed before the Constantinople upgrade. Users had to draw key pairs and then inspect the (legacy) addresses induced by the first few nonces [28]. Code-bound addresses allow to grind though seed space rather than key space. This not only facilitates the process, but also permits to out-source the grinding of vanity addresses for code accounts [4]. Those addresses can then be sold. This makes vanity addresses a commodity on the platform.

3.2 IoIs Relating to an Expansion of the Attack Surface

Honeypots (v). Morphing resurrections simplify the creation of honeypots, i. e., decoy accounts that appear vulnerable but are not. In some way, honeypots are the Ethereum version of advance fee fraud known as "491 scam" in conventional cybercrime [7]. Typically, users must send some funds to a code account (e. g., to pay fees) in order to extract greater values.

Known honeypots use barely known features of the execution environment, popular programming languages, or block explorers to trick users into misinterpreting contracts as vulnerable [39]. Resurrectable code accounts allow an attacker to deploy actually vulnerable code and then front-run the interaction of the victim with a resurrection. Simple countermeasures, which look for `SELFDESTRUCT` instructions, fail if cascaded resurrections and indirections with delegate calls are used to obscure the control flow.

Underpriced Instructions (vi). The gas price of Ethereum instructions is calibrated carefully to render resource exhaustion attacks uneconomical [12,13,2]. Resurrectable accounts offer cheaper permanent storage [6]. The idea is that instead of keeping data in the state of a code account, data can be stored as program code in a new account. Writing data this way costs roughly $2/3$ of conventional storage. (Updates are more expensive.) Reading this data can be as cheap as $1/4$ of the conventional cost if multiple words are read. While estimating an attacker's break even is beyond our scope, we note that underpriced instructions have enabled DoS attacks in the past [23,25].

4 Data Collection and Method

We parse the Ethereum blockchain until Jul 2021[9]. We extract the set of all deployed code accounts C and build the binary contract–deployer relation $R \subseteq C^2$ by iterating over all internal transactions. Let $(a, b) \in R$ denote that a is deployer of b. Our heuristic to evaluate if a contract is resurrectable is based on the this relation and some heuristics to evaluate the reachability of critical instructions.

Definition 3. *A code account $c \in C$ is **potentially resurrectable** iff this recursively defined function evaluates to true:*

$$\text{resurr}(c) = \begin{cases} \text{true}, & \textit{if } \text{create2Created}(c) \wedge \text{ sdestrReachable}(c) \\ \text{false}, & \textit{if } \text{deployer}(c) = \emptyset \ \vee \ \overline{\text{sdestrReachable}(c)} \\ \text{resurr}(\text{deployer}(c)), & \textit{otherwise.} \end{cases}$$

The deployer function is defined as: $\text{deployer}(c) = \{\ a \mid (a, c) \in R\}$. It returns either an empty set or a set of cardinality 1 from our deployer relation. The function create2Created is true if c is deployed with `CREATE2`.

The Ethereum node does not distinguish between `CREATE` and `CREATE2` in any of their public APIs. Currently, we are aware of two approaches to distinguish between the two: (a) directly instrumenting transaction executions at the level of the runtime environment level or (b) determining that `CREATE2` was used by ruling out that `CREATE` was used. We choose the latter since an instrumentation would have to rely on unstable APIs, which makes the method hard to maintain.

To rule out that code was deployed using the legacy method, we compile sets of all legacy addresses with nonces up to n_c, the current nonce of the code account. This is handled by the createAddrs function. If the address in question is not in this set, then it must be a code-bound address:

$$\text{createAddrs}(c) = \bigcup_{n=0}^{n_c} (\mathcal{H}(\text{rlp}(a_c, n)))_{96\ldots255}$$

$$\text{create2Created}(c) = \overline{\vee_{s \in \text{deployer}(c)} (a_c \in \text{createAddrs}\,(s))}.$$

The function sdestrReachable returns true if a `SELFDESTRUCT` instruction is reachable in the context of c, meaning the account can deactivate itself. Its implementation evaluates the presence of three instructions in the disassembled bytecode of the code account. First, we look for `SELFDESTRUCT` directly; if present we return true. Second, we look for `DELEGATECALL` or `CALLCODE` instructions. Both instructions preserve the context of the caller, thus making it possible that the callee can invoke `SELFDESTRUCT` in the context of the caller. If we find one of the two we also return true because we cannot rule out that the called code deactivates the account. Otherwise we return false.

[9] Block: 12817905 (2021-07-13 11:07:50 GMT+2)

To further tell if the code account is able to (a) change its state or (b) change its code upon resurrection, further non-trivial checks are needed. To rule out (a) we need to show that the initialization code does not write to storage, or more strictly, that the storage writes do not depend on the system state or user input. To rule out (b) we need to establish that the return value of the initialization code does not depend on system state or user input. Both problems are not decidable for arbitrary program code.

Here we focus on an approximation of (b) to detect morphing resurrections. We run the initialization code of every code account in question in an instrumented runtime environment to find out if it is *static* (Def. 1). Our executions stop as soon as the initialization code accesses the environment to produce its output.[10] If no access to the environment is detected, we verify that the return value corresponds to the program found on-chain.

Static initialization code is only relevant for programs with code-bound addresses. In cascading resurrections, where `CREATE` is used for deployment, the resulting program can change even if the initialization code is static. This is so because legacy addresses are not bound to the initialization code in the first place. Therefore, our complete heuristic to detect potentially morphing resurrections distinguishes between the two cases.

Definition 4. *We call a code account* ***potentially morphing*** *if it is potentially resurrectable (Def. 3) and one of the two applies: 1. it was created with* `CREATE`*; 2. its initialization code is not static (Def. 1).*

Limitations. The heuristic to detect resurrectable code accounts over-estimates the reachability of `SELFDESTRUCT` instructions, i. e., it produces false positives. Conversely, we are not aware of cases in which false negatives could occur. Therefore, from the lens of a security analysis, it always errs on the side of caution.

Our heuristics do not take into account under which circumstances the critical instructions, such as `SELFDESTRUCT`, `CREATE2`, can be reached. For instance, the community has produced examples for counterfactual instantiation without the risk of resurrections [5,3]. This is done by keeping track of already created addresses within the state of the deployer code account. Application code ensures that deployments to addresses on the list fail. Our heuristic is currently blind to such safeguards on the application layer.

Our indicator for static initialization code produces false positives if the value of a transaction is used to decide which program to return at deployment time.

5 Measurement Results

Of the total of 32 634 990 code accounts deployed after the Constantinople upgrade, 15 159 524 (47 %) are created with `CREATE2`. We proceed by discussing selected descriptive statistics for actual and then potential resurrections, before

[10] We make an exception for access to the value of a transaction, since popular compilers add checks to avoid accidental value transfer.

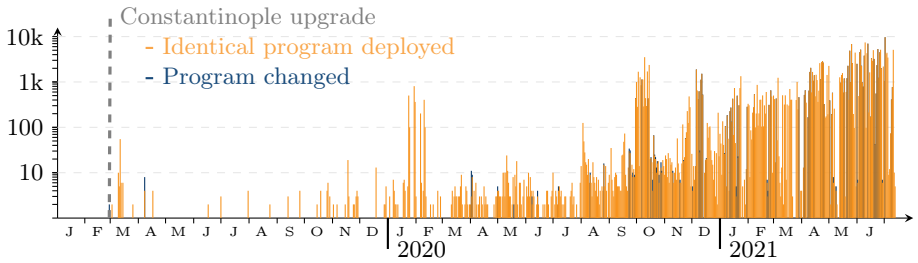


Fig. 4: Number of resurrections per day. Note the log axis.

turning to the IoIs defined in Section 3. Table 2 in the appendix reports all findings with a larger set of indicators.

5.1 Descriptive Analysis of Actual Resurrections

We find 225 032 (0.69 %[11]) resurrections since the Constantinople upgrade. They affect 73 583 distinct addresses, with an average of 3.1 resurrections per account. Figure 4 suggest that adoption took off about one year after the upgrade.

Only 41 code accounts have seen *morphing* resurrections (i. e., with code updates), of which 31 use non-static initialization code to modify the program. The remaining 10 are cascaded resurrections deployed with `CREATE`.

To better understand the use of morphing resurrections, we manually investigate the transaction behavior of the associated accounts and searched the Internet for relevant pointers.[12] Information found on Etherscan (tags and comments) and Github (known bots) suggest that 16 of the 41 are likely trading bots. We found that all but 6 of the accounts have interacted with decentralized exchanges (DEXs). Their transaction behavior resembles front-running, such as holding different tokens over time and checking the balances for different tokens regularly. 16 of the 41 accounts are still active at the time of writing (5 Oct 2021). Etherscan tags 11 of them as front-running bots. Furthermore, 25 of the accounts use vanity addresses with up to 7 leading zeros-bytes. This suggests that a tiny group of technically advanced users is responsible for the morphing resurrections in our data.

It is remarkable that a large number of resurrections happened without changing code. Figure 8a in the appendix shows the distribution of unique bytecode instances used in these resurrections. Only four programs make up more than 90 % of these resurrections. To better understand the purpose of these non-morphing but resurrecting accounts, we inspect their bytecode. The disassembly reveals that all of the four programs are gas tokens.

Gas tokens are instruments designed to hedge against gas price variations [28]. They make use of Ethereum's feature to refund gas upon freeing storage. The

[11] of total deployments after Constantinople

[12] Table 3 lists the accounts that had morphing resurrections along with selected statistics. Table 4 documents our investigations regarding the purpose of these accounts. Both tables are in the appendix.

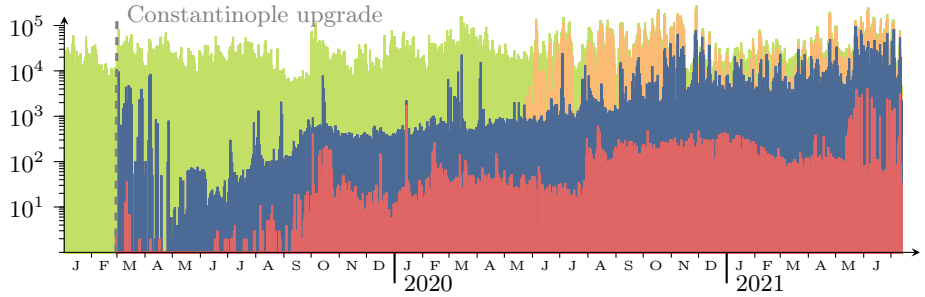


Fig. 5: Number of daily code account-creations since the Constantinople upgrade. All creations (green), code-bound addresses Def. 3 (orange), code-bound addresses without Chi-Gas-Token (blue), potentially resurrectable code-bound addresses Def. 4 (red). Note the log scale.

feature's intended purpose is to incentivize storage parsimony. However, it can be exploited to stock up gas when it is cheap by allocating excess storage. To redeem gas tokens when gas prices are high, excess storage is freed within the execution of a consuming transaction. This scheme can be used to subsidize up to 50 % of the transaction's gas cost. The use of code-bound addresses for gas tokens has been popularized by 1inch's Chi-Gas-Token [31] to reduce its own gas usage, thus making it slightly more efficient.

The resurrected gas tokens in our data were deployed by only 8 code accounts. Six of them have seen morphing resurrections and are identified as front-running bots on Etherscan (in tags or comments). This and our manual analysis suggest that all of the above mentioned deployers are front-running bots using their own gas tokens to make arbitrage trades even cheaper.

5.2 Descriptive Analysis of Potential Resurrections

Figure 5 shows the creation of code accounts since the Constantinople upgrade. The total number of new code accounts fluctuates roughly between 10^4–10^5 new instances per day.[13] About 45 % of all deployments since the upgrade (14 796 170) are potentially resurrectable code accounts, meaning that they satisfy Definition 3. Figure 9 in the appendix shows the degree distribution of the deployer relation used by this heuristic. All accounts that have seen *actual* resurrections (73 583, see Sect. 5.1) are correctly identified as *potentially resurrectable*. This increases our confidence in the validity of the heuristic indicator.

Figure 7a in the appendix shows the number of potentially resurrectable creations relative to the total number of newly created code accounts. Before May 2020, we see very little use of potentially resurrectable programs in relation to the total number of deployments. From May 2020 onwards, the share of potentially resurrectable code accounts rises sharply and stays relatively constant at around 70–80 % of all deployments.

[13] 442/12 466/29 865/38 076/279 202 (min/25%/50%/75%/max)

We can attribute a large share ($\sim 69.25\,\%$) to one system, called Chi-Gas-Token. Although its accounts satisfy our resurrection heuristic (Def. 3), they are not morphing (Def. 4) because their initialization code is always static (Def. 1).

A total of 109 195 resurrectable code accounts do *not* have static initialization code, according to our heuristic. These, plus the accounts with potential for cascaded resurrections (230 of which 159 have static initialization code) add up to 109 354 potentially problematic accounts. Recall that these accounts can potentially update their code at any time. Even worse, this possibility is not apparent even when the source code of the account is inspected. Still, the share of such accounts is small (0.74 % of all potentially resurrectable accounts) and it does not seem to increase over time (see Figure 7b).

5.3 Results on the Items of Interest

Cheaper Patching (IoI i). The very limited number (41) of accounts that had morphing resurrections suggests that only a small group of advanced users are currently aware of this possibility. Front-running bots seem to be the predominant users of morphing resurrection as they can benefit in two ways: a) it allows them to adapt their strategies easily by updating code if needed, and b) it allows them to reuse funds that are already on the account, removing the need to move funds around. Before the upgrade, this would have required a proxy, which adds a layer of indirection and thus increases communication cost. Hence, `CREATE2` mainly improves the transaction efficiency of front-running bots. We have no indication of `CREATE2` being used for cheaper patching elsewhere.

Generalized State Channels (IoI ii). This use case (as well as the next) would imply that we see a value transfer before the first deployment of the code account. To analyze that, we inspect the transaction history and parse the first four transactions of all resurrectable contracts.

For state channels, we would expect at least two participants to pre-fund the account before its creation. In addition, we expect both participants to fund the channel with amounts of the same magnitude, i. e., $\frac{1}{10} < \frac{a}{b} < 10$ where $a, b \in \mathbb{N}$. We find only 360 potentially resurrectable accounts that meet this state channel heuristic. This can mean either a) the heuristic is too restrictive (e. g., state channels use tokens for pre-funding), or b) state channels in practice do not rely on counterfactual instantiation, or c) state channels are barely used. Online information suggests that state channels are not really used on the Ethereum platform. The Github repository of Counterfactual, a project that aims to enable state channels on Ethereum, is archived and has not seen any activity for more than two years.[14] Given that state channels were the main reason for the introduction of `CREATE2` [11], this lack of adoption is surprising.

Wallet On-Boarding (IoI iii). For the purpose of on-boarding, we expect more than one transaction funding the account before it is created. As for state channels, our analysis focuses on ether and does not consider tokens. We find that

[14] https://github.com/counterfactual/monorepo, accessed: 15th Oct 21

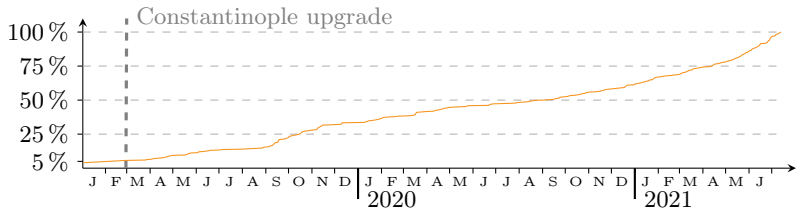


Fig. 6: Adoption of vanity addresses. Almost all of the 351 vanity addresses are created after the Constantinople upgrade. The vanity addresses are deployed by 231 accounts. Only two of them have created more than five vanity addresses. Both are immutable `CREATE2` factories [5].

696 230 of the resurrectable accounts (4.7 %) were pre-funded before deployment. This amounts to 95 % of the accounts that have been pre-funded after Constantinople.[15] To shed more light into the use of these accounts we analyze their deployers. We find 180 different deployers for all pre-funded addresses. 95 % of the accounts where created by 8 deployers only. We could identify 82 % of the deployed accounts as either wallets or forwarder accounts. Forwarder accounts are used to create fresh deposit addresses [27,18]. In summary, we observe that wallet on-boarding is indeed a practical use case of `CREATE2` for its predictable addresses, but this use case does not need the feature to update code.

Vanity Addresses (IoI iv). Figure 6 shows that most vanity addresses are deployed after the Constantinople upgrade.

The upgrade in principle enables the commodification of vanity addresses for code accounts. For this, we would expect a small number of suppliers specialized on grinding vanity addresses and selling them to interested parties by deploying code on their customers' behalf. (Recall that `CREATE2` still binds the derived address to the deployer. Therefore, only the supplier can create a contract at a derived address.) So far, we do not see any indication of trade with vanity addresses. Only one code account deployed a significant share (84 of 351) of vanity addresses.[16] It implements a code account that facilitates deployments using `CREATE2` without the risk of resurrections [5], but it does not include any trading functionality. Moreover, the front-running bot that brings their own version of gas tokens (see Sect. 5.1) also uses vanity addresses. This increases gas efficiency at deployment time because addresses are stored in bytecode without leading zeros.

Honeypots, IoI v. Since we see so few mutating resurrections, it is safe to state that `CREATE2` is not used in honeypots at a large scale.

Underpriced Instructions, IoI vi. It is generally hard to tell if code accounts are repurposed as cheap long-term data storage. Analyzing this would require low-level instrumentation of the execution environment to measure the use of

[15] In total we have seen 731 268 pre-funded accounts after Constantinople. As expected, almost all of them (729 577) were deployed via `CREATE2`.

[16] `0x0000000000ffe8b47b3e2130213b802212439497`

`EXTCODECOPY` and the values derived from it in computations. What we can conclude from our measurement, however, is that resurrections are currently not widely used as a cheap substitute for *updateable* storage, as this would involve morphing resurrections. To evaluate the potential of contracts being used as write-once storage, we make the assumption that such a storage code account would never receive or send any messages. Data is read using the `EXTCODECOPY` instruction only. We find 697 760 code accounts created using *`CREATE2`* that match this assumption.[17] Most of them (82 %) are Chi-Gas-Tokens, leaving us with an upper bound of 125 692 potential write-once storage accounts.

6 Related Work

This work speaks to how users can verify the immutability of smart contracts, and the tension between immutable code and the ability to correct bugs. It also contributes to the nascent literature measuring the impact of platform upgrades.

Immutability. Fröwis et al. [20] study the immutability of control flow with static program analysis. Using heuristics, they estimate that 40% of the code accounts on Ethereum host program code that could change its control flow by updating dynamic references. Code updates were not on the horizon as the study predates the Constantinople upgrade by several years. Zhou et al. [41] present Erays, an Ethereum reverse engineering tool that lifts EVM bytecode to human-readable pseudo-code. The human-readable representation simplifies manual inspection of smart contracts and thus the identification of mutable code paths. Di Angelo et al. [17] review unintended use-cases of initialization code on Ethereum. They argue that a lack of understanding of the role of initialization code opens opportunities for deception and fraud in scenarios like token harvesting and gambling. In particular, initialization code can be used to circumvent automated checks to tell code accounts and externally owned accounts apart.

Patching. Numerous academic works concern the identification of vulnerabilities in smart contracts [22,16]. Whereas, only a couple of publications discuss techniques to fix bugs once discovered. We focus on the latter. Rodler et al. [37] present `EVMPatch`, a tool that uses bytecode rewriting to automate the creation of updateable smart contracts with the transparent proxy pattern. It combines bytecode analysis with regression testing—partly based on the transaction history—to avoid breaks during the update. Azzopardi et al. [8] explore the use of runtime verification to recover from bugs in code accounts. They present `ContractLarva`, a monitoring framework that allows to encode execution invariants and recovery strategies using a domain specific language. Colombo et al. [15] extend the work of Azzopardi and discuss recovery approaches in the case of violated execution invariants. They describe how invariant-preserving updates

[17] To reduce the bias by contracts created towards the end of our observation period, we only considered code account until two month before the end of our measurement period. Without this, cutoff we find 3 158 545 matching code accounts.

could be implemented using the proxy pattern. Dickerson et al. [19] take another approach. They propose proof-carrying code to allow for invariant-preserving updates. Every program deployed comes with a proof that some important invariant is never violated. In the event of a patch, the runtime system ensures that the new code preserves the invariants specified in the original code.

All of the works mentioned above rely on the proxy pattern to enable code updates. The works of Rodler and Dickerson would benefit from the cheaper patch mechanism provided by `CREATE2`. The approach of Azzopardi and Colombo can only partly benefit from `CREATE2`; they reuse the proxy to encode the behavioral invariants the contract must preserve, thus adapting this to `CREATE2` does not immediately avoid expensive indirections.

Platform Upgrades. We are aware of three measurement studies that directly evaluate the effects of platform upgrades. Perez et al. [2] and Chen et al. [13] measure underpriced instructions after the upgrade including EIP-150, which increased gas prices to avoid DoS attacks. Recently, Reijsbergen et al. [36] evaluate the effects of the changes in the calculation of transaction fees introduced by EIP-1559. We deem this under-researched and believe that the governance of platforms such as Ethereum should stand on solid empirical foundations.

7 Conclusion

This work describes the impact and measures the adoption of `CREATE2`, a new EVM instruction that makes address assignment more predictable. Our measurement is motivated by a side effect of `CREATE2` with security implications, namely the ability to patch code accounts in place. We have proposed a novel heuristic indicator to identify `CREATE2`-patchable code accounts, and applied it to the the relevant two years of Ethereum blockchain data.

The data shows that `CREATE2` has become the dominant form of deployment since May 2020. But the dominance is largely driven by gas tokens using `CREATE2` to increase their efficiency. As expected, wallet and forwarder contracts have also adopted `CREATE2`. They benefit from the intended use case of `CREATE2`, *counterfactual instantiation*, i. e., contracts that can be funded before they are deployed. Interestingly, we do not find any indication of state channels using `CREATE2`, despite this was the proclaimed reason for the addition of `CREATE2`.

Patching via `CREATE2` has seen very little use since the upgrade. Mainly the infrastructure of front-running seems to take advantage of it. A manual analysis of all relevant accounts lets us conclude that this new and opaque form of code patching is—so far—not used maliciously to defraud users. Still, according to our heuristic there are more than 100k accounts with the potential to change their code in the future. The community is well advised to keep watching this.

In summary, we conjecture that the main reason for the limited use of `CREATE2` for patching is a lack of awareness. Moreover, none of the uses cases we discussed, or that were discussed in the proposal of `CREATE2`, use the patching feature. This feature increases the attack surface without need, and offers limited benefit to a few experienced users only.

Acknowledgments

We would like to thank Patrik Keller and Bernhard Haslhofer for their valuable feedback. This work has received funding from the Austrian Research Promotion Agency (FFG) and the Austrian Security Research Programme (KIRAS).

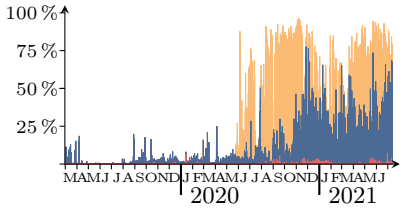
References

1. EVM - implement EIP 1014: Skinny CREATE2 #1165. https://github.com/trailofbits/manticore/issues/1165, [Online; accessed 28 Oct 2021]
2. 0002, D.P., Livshits, B.: Broken Metre: Attacking Resource Metering in EVM. In: 27th Annual Network and Distributed System Security Symposium, NDSS. The Internet Society (2020)
3. 0age: Metamorphic. https://github.com/0age/metamorphic, [Online; accessed 07 April 2021]
4. 0age: On Efficient Ethereum Addresses. https://medium.com/coinmonks/on-efficient-ethereum-addresses-3fef0596e263 (2018), [Online; accessed 20 May 2021]
5. 0age: Etherscan CREATE2SafeDeploy. https://etherscan.io/address/0x5df4c8e56fe3a95f98ce3d1935abd1b187525915/ (2019), [Online; accessed 07 April 2021]
6. 0age: On Efficient Ethereum Storage. https://medium.com/coinmonks/on-efficient-ethereum-storage-c76869591add (Mar 2019), [Online; accessed 25 Mai 2021]
7. Anderson, R., Barton, C., Böhme, R., Ganan, C., Grasso, T., Levi, M., Moore, T., Vasek, M.: Measuring the Changing Cost of Cybercrime. In: Workshop on the Economics of Information Security (WEIS). Harvard University, Cambridge, MA (2019)
8. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond. In: International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 11237, pp. 113–137. Springer (2018)
9. Böhme, R., Eckey, L., Moore, T., Narula, N., Ruffing, T., Zohar, A.: Responsible Vulnerability Disclosure in Cryptocurrencies. Communications of the ACM 63(10), 62–71 (2020)
10. Buterin, V.: Prevent overwriting contracts #684. https://github.com/ethereum/EIPs/issues/684 (2017), [Online; accessed 07 April 2021]
11. Buterin, V.: EIP 1014: Skinny Create2. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1014.md (2018), [Online; accessed 07 April 2021]
12. Chen, T., Li, X., Luo, X., Zhang, X.: Under-Optimized Smart Contracts Devour Your Money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 442–446. IEEE (2017)
13. Chen, T., Li, X., Wang, Y., Chen, J., Li, Z., Luo, X., Au, M.H., Zhang, X.: An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks. In: International Conference on Information Security Practice and Experience. Lecture Notes in Computer Science, vol. 10701, pp. 3–24. Springer (2017)

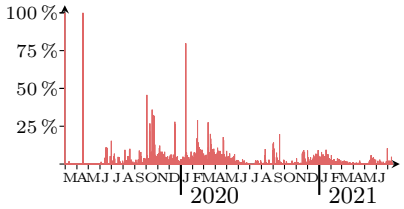
14. Coleman, J., Horne, L., Xuanji, L.: Counterfactual: Generalized State Channels. https://l4.ventures/papers/statechannels.pdf (2018), [Online; accessed 07 April 2021]
15. Colombo, C., Ellul, J., Pace, G.J.: Contracts over Smart Contracts: Recovering from Violations Dynamically. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. Lecture Notes in Computer Science, vol. 11247, pp. 300–315. Springer International Publishing, Cham (2018)
16. Di Angelo, M., Salzer, G.: A Survey of Tools for Analyzing Ethereum Smart Contracts. In: 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON). pp. 69–78. IEEE (2019)
17. Di Angelo, M., Salzer, G.: Collateral Use of Deployment Code for Smart Contracts in Ethereum. In: 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5. IEEE (2019)
18. Di Angelo, M., Salzer, G.: Characteristics of Wallet Contracts on Ethereum. In: 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 232–239. IEEE (2020)
19. Dickerson, T., Gazzillo, P., Herlihy, M., Saraph, V., Koskinen, E.: Proof-Carrying Smart Contracts. In: Zohar, A., Eyal, I., Teague, V., Clark, J., Bracciali, A., Pintore, F., Sala, M. (eds.) Financial Cryptography and Data Security, WTSC Workshop. Lecture Notes in Computer Science, vol. 10958, pp. 325–338. Springer Berlin Heidelberg (2019)
20. Fröwis, M., Böhme, R.: In Code We Trust? Measuring the Control Flow Immutability of All Smart Contracts Deployed on Ethereum. In: Garcia-Alfaro, J., Navarro-Arribas, G., Hartenstein, H., Herrera-Joancomartí, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology, ESORICS 2017 International Workshops. Lecture Notes in Computer Science, vol. 10436, pp. 357–372. Springer, Cham (2017)
21. Gabriel Barros, P.G.: EIP-1822: Universal Upgradeable Proxy Standard (UUPS). https://eips.ethereum.org/EIPS/eip-1822 (2019), [Online; accessed 07 April 2021]
22. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In: International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 11561, pp. 51–78. Springer (2018)
23. Hertig, A.: So, Ethereum's Blockchain is Still Under Attack.... http://www.coindesk.com/so-ethereums-blockchain-is-still-under-attack/ (Oct 2016), [Online; accessed 18 June 2017]
24. Isaac Ardis, W.T.: ECIP 1056: Agharta EVM and Protocol Upgrades. https://ethereumclassic.org/blog/2020-01-11-agharta-hard-fork-upgrade (Nov 2020), [Online; accessed 25 Mai 2021]
25. Jameson, H.: FAQ: Upcoming Ethereum Hard Fork. https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/ (Oct 2016), [Online; accessed 18 June 2017]
26. Jorge Izquierdo, M.A.: EIP-897: ERC DelegateProxy. https://eips.ethereum.org/EIPS/eip-897 (2018), [Online; accessed 07 April 2021]
27. Joveski, B.: USDC payment processing in Coinbase Commerce. https://blog.coinbase.com/usdc-payment-processing-in-coinbase-commerce-b1af1c82fb0 (Aug 2019), [Online; accessed 18 November 2021]
28. Lorenz Breidenbach, Phil Daian, F.T.: GasToken.io - Cheaper Ethereum transactions, today. https://gastoken.io/#GST2, [Online; accessed 19 Oct 2021]

29. Maurelian: Newsletter 16 — CREATE2 FAQ. https://consensys.net/diligence/blog/2019/02/smart-contract-security-newsletter-16-create2-faq/ (2019), [Online; accessed 08 Mai 2021]
30. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1186–1189. IEEE (2019)
31. 1inch Network: 1inch introduces Chi Gastoken. https://blog.1inch.io/1inch-introduces-chi-gastoken-d0bd5bb0f92b, [Online; accessed 05 Mai 2021]
32. OpenZeppelin: OpenZeppelin Proxy Contract Implementations. https://docs.openzeppelin.com/contracts/4.x/api/proxy, [Online; accessed 07 Nov 2021]
33. Palladino, S.: EIP-1967: Standard Proxy Storage Slots. https://eips.ethereum.org/EIPS/eip-1967 (2019), [Online; accessed 07 April 2021]
34. Peter Murray, Nate Welch, J.M.: EIP-1167: Minimal Proxy Contract. https://eips.ethereum.org/EIPS/eip-1167 (2018), [Online; accessed 07 April 2021]
35. rajeevgopalakrishna: Potential security implications of CREATE2? (EIP-1014). https://ethereum-magicians.org/t/potential-security-implications-of-create2-eip-1014/2614 (2019), [Online; accessed 08 April 2021]
36. Reijsbergen, D., Sridhar, S., Monnot, B., Leonardos, S., Skoulakis, S., Piliouras, G.: Transaction Fees on a Honeymoon: Ethereum's EIP-1559 One Month Later. arXiv preprint arXiv:2110.04753 (2021)
37. Rodler, M., Li, W., Karame, G.O., Davi, L.: EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts. In: 30th USENIX Security Symposium. USENIX Association (2021)
38. (((Swende))), M.H.: Testing awareness levels here. After Constantinople, can contracts that you interact suddenly change code, in-place? https://twitter.com/mhswende/status/1093596010545336320 (2019), [Online; accessed 06 Oct 2021]
39. Torres, C.F., Steichen, M., et al.: The Art of the Scam: Demystifying Honeypots in Ethereum Smart Contracts. In: 28th USENIX Security Symposium. pp. 1591–1607. USENIX Association (2019)
40. Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger (Petersburg revision). https://ethereum.github.io/yellowpaper/paper.pdf (2021), [Online; accessed 07 April 2021]
41. Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., Bailey, M.: Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In: 27th USENIX Security Symposium. p. 1371–1385. USENIX Association (2018)

A Supplemental Figures and Tables

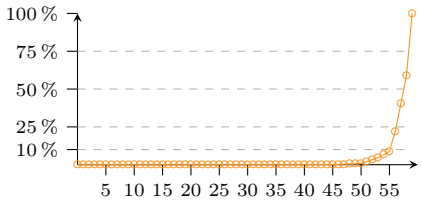


(a) Daily creations of potentially resurrectable code accounts, relative to the total number of creations and the number of creations of code-bound addresses.

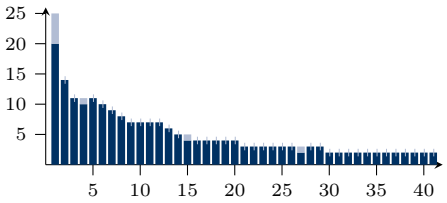


(b) Percentage of new potentially resurrectable account creations (red) in relation to total resurrectable code, without Chi-Gas-Token.

Fig. 7: New potentially resurrectable code accounts per day



(a) 60 bytecode instances used by 73 554 resurrecting accounts that never changed their code (over a total of 224 830 resurrections). Only four bytecode instances appear in 90 % of the deployments.



(b) 189 bytecode instances used in 202 morphing resurrections on 41 accounts. The light blue parts show the code instances seen more than once. Almost all morphing resurrections deploy fresh code.

Fig. 8: Bytecode instances used in resurrections; redeployment of identical code (left) and morphing resurrections (right).

Figure 9 shows the cumulative distribution of out-degrees of the deployer-relation.[18] Only 189 919 (0.41 %) accounts in our relation have no deployer. These result from deployments from externally owned accounts. Hence, almost all code accounts were created by another code account.

[18] The in-degree distribution is binary since every code account has exactly one deployer. Only externally owned accounts have no deployer.

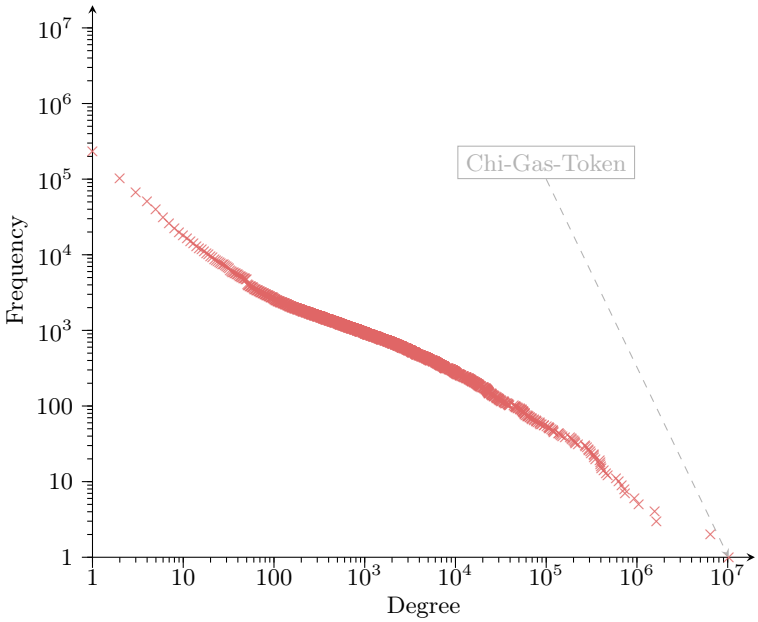


Fig. 9: The Ethereum deployer relation. Only 5 code accounts deployed more than 1 million other code accounts. The biggest contributors are the Chi-Gas-Token (≈ 10m), Gas Token.io: GST2 (≈ 6.4m), MevBot (≈ 1.6m), Bitrex: Controller (≈ 1.6m), MMM BSC (≈ 1m). MevBot is one of the contracts that have seen morphing resurrections.

Table 1: Distribution of actual resurrections per account

Concept	Statistic		ref
Nr. of resurrections per account	mean	3.1	Sec. 5.1
	min	2.0	–
	25 %	2.0	–
	50 %	2.0	–
	75 %	3.0	–
	max	645.0	–
Nr. of morphing resurrection per account	mean	5.2	–
	min	2.0	–
	25 %	2.0	–
	50 %	3.0	–
	75 %	7.0	–
	max	25.0	–

Table 2: Dataset summary

	Concept	#	Share of*	Reference
1	Highest block	12 817 905	– –	Sec. 4
2	Total code accounts created	45 666 538	– –	–
3	Creations after Constantinople	32 634 990	71.5 % 2	Sec. 5
4	Pre-funded before deployment	731 268	2.2 % 3	Sec. 5.1, IoI iii
5	State channel pre-funding	7231	– –	IoI ii
6	Total *CREATE2* creations	15 159 524	46.5 % 3	Sec. 5
7	Pre-funded before deployment	729 577	99.69 % 4	Sec. 5.1, IoI iii
8	State channel pre-funding	7173	99.2 % 5	IoI ii
9	Vanity addresses	351	– –	IoI iv, Def. 2
	Observed Resurrections			
10	**Total resurrections**	225 032	1.5 % 6	Sec. 5.1
11	Accounts with resurrections	73 583	– –	Sec. 5.1
12	Unique programs	247	– –	–
13	Unique initialization codes	549	– –	–
14	Unique deployers	123	– –	–
15	Vanity addresses	27	7.7 % 9	IoI iv, Def. 2
16	Pre-funded before deployment	360	0.05 % 4	Sec. 5.1, IoI iii
17	State channel pre-funding	2	0.03 % 5	IoI ii
18	**Morphing resurrections**	202	0.1 % 10	IoI v
19	Accounts with resurrections	41	0.05 % 11	Sec. 5.1, IoI i
20	Unique programs	189	76.5 % 12	–
21	Unique initialization codes	18	3.3 % 13	–
22	Unique deployers	30	24.4 % 14	–
23	Vanity addresses	25	92.6 % 15	IoI iv, Def. 2
24	Pre-funded before deployment	1	– –	Sec. 5.1, IoI iii
25	State channel pre-funding	0	– –	IoI ii
	Potential for Resurrections			
26	**Potentially resurrectable accounts**	14 796 170	45.3 % 3	Sec. 5.2, Def. 3
27	Unique programs	11 024	– –	
28	Unique initialization codes	58 295	– –	–
29	Unique deployers	1218	– –	–
30	Vanity addresses	66	18.8 % 9	Sec. 5.1, IoI iv, Def. 2
31	Pre-funded before deployment	696 230	95.2 % 4	Sec. 5.1, IoI iii
32	State channel pre-funding	6323	87.4 % 5	Sec. 5.1, IoI ii
33	**Without Chi-Gas-Token**	4 549 820	30.7 % 26	–
34	Unique programs	11 023	99.99 % 27	–
35	Unique initialization codes	58 294	100 % 28	–
36	Unique deployers	1212	99.5 % 29	–
37	Vanity addresses	66	100 % 30	Sec. 5.1, IoI iv, Def. 2
38	Pre-funded before deployment	696 228	99.99 % 31	IoI iii
39	State channel pre-funding	6323	100 % 32	IoI ii
40	**Potentially Morphing**	109 354	2.4 % 34	Sec. 5.1, Def. 4
41	Unique programs	286	2.6 % 35	–
42	Unique initialization codes	29 739	51 % 36	–
43	Unique deployers	327	27 % 37	–
44	Vanity addresses	46	69.7 % 38	IoI iv, Def. 2
45	Pre-funded before deployment	5970	– –	IoI iii
46	State channel pre-funding	248	– –	IoI ii

* **of** refers to the row number used as base to calculate the percentage.

Table 3: Accounts with morphing resurrection until mid Jul 2021. USD values based on current ETH Price (Coinbase midpoint price 5th Oct 21).

Address	Deployed	First seen	Last seen	Calls in	Calls out	Nr deploy.	Tsd. USD in	Nr redeploy.	Cascaded	Static IC
0x45503c34ee615c4ed5fe071fc6ccee06c049b1e3	Apr 03 20	Apr 02 20	Apr 03 20	20	31	0	0.4	9	○	○
0x000000000000006f6502b7f2bbac8c30a3f67e9a	Jun 21 21	May 01 20	Jul 13 21	1706881	2154591	210025	4511394.6	11	○	○
0x000000009b988fbecfd83c55252f78592e609648	Aug 11 20	Aug 11 20	Aug 11 20	3	3	0	0.0	2	○	○
0xd1c300000000b961df238700ef00600097000049	Nov 24 20	Aug 21 20	Nov 24 20	31819	109464	0	0.0	2	○	●
0x00f8d52a00f60094ebf903ce0000defb00f20000	Aug 26 20	Aug 26 20	Aug 28 20	50	124	0	18.7	2	○	○
0x000000000000740a22fa209cf6806d38f7605385	Sep 29 20	Sep 23 20	Oct 01 20	12083	29775	2729	4592.8	6	○	○
0x00000000005af5bf8581e5143248bcf395abb892	Sep 24 20	Sep 24 20	Sep 24 20	19	37	0	0.0	2	○	○
0x00000000002bde777710c370e08fc83d61b2b8e1	Oct 26 20	Sep 24 20	Nov 27 20	51592	131551	18412	0.0	7	○	○
0x000000000000084e91743124a982076c59f10084	May 09 21	Sep 30 20	Jul 13 21	306669	680114	91126	93585.6	11	○	○
0x000000000000a32dc5dd625c107898a1c72ad34a	Oct 21 20	Oct 07 20	Nov 27 20	3147	7938	1258	0.0	7	○	○
0x00000000007e35dff0153a18de63c4f58f04fd1c	Oct 17 20	Oct 17 20	Oct 18 20	74	77	0	11.3	2	○	○
0x0000000000159bc46c893771699455745ba335c8	Oct 23 20	Oct 22 20	Oct 23 20	19	120	106	1.1	3	○	○
0x000000000007f150bd6f54c40a34d7c3d5e9f56	Jun 23 21	Oct 23 20	Jul 13 21	3866901	17286031	1647068	13194058.0	25	○	○
0x0000000000000eb4ec62758aae93400b3e5f7f18	May 05 21	Nov 10 20	Jul 13 21	149167	943153	3254	224884.2	3	○	○
0x0000000000438f975cbde76d5fdd1aa76be46577	Dec 10 20	Nov 22 20	Apr 21 21	14461	183112	0	5555.6	3	○	○
0x009eede3040800db000064ab00bd00f1a3000b38	Jan 22 21	Nov 26 20	Jun 15 21	798	2774	0	138294.0	2	○	○
0x49307d775728daf1d4736ab762de0cefd035e323	May 21 21	Dec 06 20	Jul 13 21	93570	218701	29559	13773.4	8	●	●
0x1d6e8bac6ea3730825bde4b005ed7b2b39a2932d	Jul 10 21	Dec 06 20	Jul 13 21	281995	659645	79444	49336.1	14	●	●
0x1b3006d57d5d9653a4ecaea6cfb0fd72a1537cc9	Mar 04 21	Dec 12 20	Jul 11 21	60693	164671	62131	3316.4	3	●	●
0x577a32aa9c40cf4266e49fc1e44c749c356309bd	Apr 24 21	Dec 12 20	Apr 24 21	32106	57804	6149	2752.7	4	●	●
0x00000000000b69ec332f49b7c4d2b101f93c3bed	May 14 21	Jan 07 21	May 14 21	97	141	0	36.6	3	○	○
0x000000005736775feb0c8568e7dee77222a26880	Jun 28 21	Jan 09 21	Jul 13 21	60853	139957	35553	7050.9	11	○	○
0x000000000000f095b1b50de78b7a6bc4697b2927	May 04 21	Jan 12 21	Jul 12 21	1902	6309	579	17.2	7	○	○
0x4bd372cd9bedd45e0e23c769bb1879dd055a32f2	Mar 04 21	Jan 15 21	Jun 01 21	141	1030	500	8.7	2	●	●
0x00000000751edf02c727ea9d0c1c8e64f169e343	Jan 31 21	Jan 19 21	Apr 14 21	15843	27097	2951	0.0	4	○	○
0x000000000000d736bd9b915d8fe98e9c507df85a	Jan 21 21	Jan 20 21	Mar 26 21	5700	20659	1826	8.6	2	○	○
0xf3c62891b48c5152dca70e0da62f7e1d8c2cc5bb	Feb 26 21	Feb 23 21	Feb 26 21	190	139	31	0.0	5	○	○
0xff0cca1818ddc0d66b53a96a9aa8bda9043e1437	Mar 29 21	Mar 29 21	Apr 08 21	163	6	0	0.0	7	●	●
0x000000000003c42172c0bdd69d6afb6b0bf4b488	Apr 26 21	Apr 22 21	May 14 21	1240	3546	1178	16.8	4	○	○
0xc79eb641599e51930acd560c9dd60d81ca4fb831	Jun 06 21	Apr 23 21	Jul 13 21	665	3771	0	1151.3	2	○	○
0x00000000b7ca7e12dcc72290d1fe47b2ef14c607	May 06 21	Apr 28 21	Jul 13 21	125222	283691	101796	4991.2	2	○	○
0x00000000003b3cc22af3ae1eac0440bcee416b40	Jul 02 21	May 21 21	Jul 13 21	120562	261578	86112	14092.5	10	○	○
0x0000000099cb7fc48a935bceb9f05bbae54e8987	Jun 26 21	May 24 21	Jul 13 21	68455	153976	42912	5090.3	3	○	○
0x21a3d0b8289de546402fdfff2b1c860975cda090	Jun 07 21	May 25 21	Jul 13 21	593	1716	117	4.6	3	○	○
0xc65433d2a598c323e04fd143566b08609af008dc	Jun 06 21	Jun 01 21	Jun 08 21	3495	10088	1700	1064.1	4	●	●
0x000000000035b5e5ad9019092c665357240f594e	Jul 09 21	Jun 02 21	Jul 13 21	50540	114525	33312	1863.5	3	○	○
0x55eb58655f8202ff839487886fedba2a1eb7b2d7	Jun 06 21	Jun 06 21	Jun 08 21	34	68	0	1470.1	3	●	●
0x0000000000a680073b2eee93abeb4af4d4a096f8	Jul 09 21	Jun 06 21	Jul 13 21	1268	7062	3040	48.1	4	○	○
0x00000000032962b51589768828ad878876299e14	Jun 20 21	Jun 09 21	Jul 13 21	7079	17182	6240	310.9	2	○	○
0x000000009042b40070c8f83bff1293005dc25257	Feb 28 19	Feb 28 19	Feb 28 19	4	3	0	0.0	2	○	○
0x7f000006128132800740006 00f710002bdbaa9c	May 06 21	Apr 08 19	May 06 21	14	7	0	70.7	5	●	●

Table 4: Analysis of morphing accounts; Balances from Etherscan on 5th October 21.

Address	Active	DEX	Known Bot	Bot Comment	Bot tag	Comments	Balance USD
0x000000000000006f6502b7f2bbac8c30a3f67e9a	●	●	●	●	○	by eth	48m
0x0000000000007f150bd6f54c40a34d7c3d5e9f56	●	●	●	●	●	third largest contract deployer	3.8m
0x0000000032962b51589768828ad878876299e14	●	●	○	○	●	Miner are top receivers	2.5m
0x55eb58655f8202ff839487886fedba2a1eb7b2d7	●	●	○	○	○	—	833k
0x0000000000000eb4ec62758aae93400b3e5f7f18	●	●	○	○	○	comments suggest Nice Hash but transactions look a like Front-running bot	736k
0x0000000099cb7fc48a935bceb9f05bbae54e8987	●	●	○	●	●	—	219k
0x000000000035b5e5ad9019092c665357240f594e	●	●	○	●	●	—	100k
0x000000000000084e91743124a982076C59f10084	●	●	●	●	●	—	68k
0x0000000000a680073b2eee93abeb4af4d4a096f8	●	●	○	○	○	—	0
0x49307d775728daf1d4736ab762de0cefd035e323	●	●	○	●	○	—	0
0x1d6e8bac6ea3730825bde4b005ed7b2b39a2932d	●	●	○	●	○	—	0
0x00000000000b69ec332f49b7c4d2b101f93c3bed	●	●	○	○	○	—	0
0x000000005736775feb0c8568e7dee77222a26880	●	●	○	●	●	—	0
0x00000000003b3cc22af3ae1eac0440bcee416b40	●	●	○	●	●	—	0
0x00000000b7ca7e12dcc72290d1fe47b2ef14c607	●	●	○	●	●	—	0
0x000000000000d736bd9b915d8fe98e9c507df85a	●	●	○	○	○	—	0
0x45503c34ee615c4ed5fe071fc6ccee06c049b1e3	○	○	○	○	○	—	350
0x000000000000740a22fa209cf68064d38f7605385	○	●	○	●	○	—	0
0x00000000002bde777710c370e08fc83d61b2b8e1	○	●	●	●	○	—	0
0x000000000000a32dc5dd625c107898a1c72ad34a	○	●	●	○	○	—	0
0x0000000000159bc46c893771699455745ba335c8	○	○	○	○	○	—	0
0x0000000000438f975cbde76d5fdd1aa76be46577	○	●	○	○	○	—	0
0x1b3006d57d5d9653a4ecaea6cfb0fd72a1537cc9	○	●	○	●	○	—	0
0x577a32aa9c40cf4266e49fc1e44c749c356309bd	○	●	○	○	○	—	344
0x000000000000f095b1b50de78b7a6bc4697b2927	○	●	○	○	○	—	7k
0x00000000751edf02c727ea9d0c1c8e64f169e343	○	●	○	○	○	—	0
0xf3c62891b48c5152dca70e0da62f7e1d8c2cc5bb	○	○	○	○	○	used only USDT	0
0xff0cca1818ddc0d66b53a96a9aa8bda9043e1437	○	●	○	○	○	Yearn Link Token see Etherscan Token Tracker, invoked by Ethermine: MEV Sender	0
0x000000000003c42172c0bdd69d6afb6b0bf4b488	○	●	○	○	○	—	0
0x21a3d0b8289de546402fdfff2b1c860975cda090	○	●	○	○	○	—	0
0xc65433d2a598c323e04fd143566b08609af008dc	○	●	○	○	○	—	0
0x7f0000006128132800740006 00f710002bdbaa9c	○	○	○	○	○	—	0
0x000000009b988fbecfd83c55252f78592e609648	○	○	○	○	○	—	0
0xd1c300000000b961df238700ef00600097000049	○	●	●	○	○	comments suggest Nice Hash but transactions look a like Front-running bot	0
0x00f8d52a00f60094ebf903ce0000defb00f20000	○	●	○	○	○	—	0
0x00000000005af5bf8581e5143248bcf395abb892	○	●	○	○	○	—	0
0x000000000007e35dff0153a18de63c4f58f04fd1c	○	●	○	○	○	—	0
0x009eede30408 00db000064ab00bd00f1a3000b38	○	●	○	○	○	—	0
0x4bd372cd9bedd45e0e23c769bb1879dd055a32f2	○	●	○	○	○	—	0
0xc79eb641599e51930acd560c9dd60d81ca4fb831	○	●	○	○	○	—	0
0x000000009042b40070c8f83bff1293005dc25257	○	○	○	○	○	—	0

Active: Account was active in the last 30 days. **DEX:** Account used DEX swap function in a transaction.
Known Bot: Address can be found in known bot list, https://github.com/flashbots/mev-inspect-rs/blob/master/src/addresses.rs.
Bot Comment: Etherscan comments suggest front-running, **Bot Tag:** Etherscan tag suggest front-running.
Balances are the sum of token an ETH balances.