Constant-Gas Programming for Confidential Smart Contracts

Meng Xu
University of Waterloo
meng.xu.cs@uwaterloo.ca

Abstract

Confidential smart contract execution try to keep calldata and contract state secret while preserving the familiar execution model of Ethereum-like blockchains. This combination, however, unexpectedly creates a side channel that constant-time programming does not close. In the EVM, storage metering distinguishes whether a write sets a zero slot, clears a non-zero slot, writes back the same value, or updates a non-zero value to another. Thus a contract can execute the same bytecode and touch the same storage slots, yet still leak secrets through the semantic class of its storage updates.

This paper presents *constant-gas programming* for confidential smart contracts. We formalize the attacker as a marker-aware write oracle M^+ and define constant-gas security as update-oblivious noninterference over EVM-style traces. We then give concrete evidence that Oasis Sapphire localnet is not safe against M^+ adversaries. To defend, we build Coga, a proof-carrying toolchain for a subset of Solidity that compiles and executes confidential smart contracts with proof artifacts showing that the execution is M^+-safe. Our evaluation shows that Coga is practical for Solidity contracts involving scalars and small and bounded mappings, and can optionally use Path ORAM for contracts involving large mappings.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**; *Software security engineering*; *Formal security models.*

Keywords

confidential smart contracts, trusted execution environments, side channels, constant-time programming, gas metering, ORAM

1 Introduction

Timing attack is a side-channel attack that poses a serious threat to otherwise secure software. Such attacks allow adversaries to gain information about secret values by monitoring an execution through observable channels, such as total execution time, microarchitectural states, memory accesses, or filesystem activity [1, 8, 10, 11]. As a countermeasure, security practitioners often rely on *constant-time programming*. Informally, a program is constant-time if its control flow and memory-address trace do not depend on secrets [1, 2].

On the first look, constant-time programming appears to be a natural fit for confidential smart contracts. Platforms such as Oasis Sapphire run EVM-compatible smart contracts with encrypted calldata and confidential contract state inside a trusted execution environment (TEE) such as an Intel SGX Enclave, while retaining the programming model expected by Solidity developers [13, 14]. If secret-dependent branches and secret-dependent storage keys are removed, one may hope that confidential execution is side-channel resistant even on an untrusted host platform.

Unfortunately, this hope misses a Solidity-specific but conceptually general problem. EVM storage gas metering is value sensitive. The cost of an `SSTORE` depends not only on which storage slot is written, but also on how the value in the slot changes. In particular, EIP-2200 and later gas reforms distinguish cases such as zero-to-nonzero, nonzero-to-zero, same-value writes, dirty-slot writes, and cold or warm access states [4, 5, 21]. A confidential EVM may encrypt the value itself, pagging to a fixed gas cost may hide the most obvious public-observable leak, but the execution traces observable on untrusted host can still expose the *semantic class* of the update.

This is exactly the blind spot. A conventional constant-time observer sees which location is touched. It does not see whether the location is set (zero to nonzero), cleared (nonzero to zero), or stayed the same. But confidential smart contracts execute on a platform where such distinctions are part of the resource model, and where the host running the confidential runtime can often see more than the public transaction receipt. As a result, a contract can be constant-time but still leak private information.

Figure 1 gives the minimal example. The loop scans all token slots in a deterministic order, so the location trace does not reveal which token matched. Nevertheless, the final assignment to `last` can reveal whether authentication succeeded, whether the same token was used as before, or whether the result changed from one nonzero token index to another. The leak is not caused by a secret-dependent branch or memory indexing. It is caused by storage-update semantics.

```
contract Authentication {
  bytes32[256] public TOKENS;
  uint8 private last = 0;

  function auth(uint256 token) public returns (bool) {
    bytes32 h = keccak256(abi.encode(token));
    uint8 r = 0;
    for (uint8 i = 0; i < TOKENS.length; i++) {
      r |= (TOKENS[i] == h) ? i : 0;
    }
    last = r;
    return r != 0;
  }
}
```

Figure 1: A constant-time Solidity smart contract that is not constant-gas. The loop length and accessed `TOKENS` slots are fixed, but `last = r` writes to a global slot and leaks whether the authentication result is zero, nonzero, unchanged, or changed.

In this paper, we argue that confidential smart contracts need a stronger discipline: *constant-gas programming*. The name comes from smart-contract execution, where instruction execution, memory expansion, storage access, and storage updates are reflected in gas or gas-like accounting. The scientific core, however, is broader and cleaner: constant-gas is update-obliviousness specialized to EVM-style persistent state. A constant-gas contract must have a

public control schedule, a public address schedule, and a public update-class schedule. Gas padding, a common practice by popular confidential EVM solutions (e.g., Oasis Sapphire), does not make the execution update-oblivious, as demonstrated in §4.

More specifically, this paper makes four contributions.

- *A precise constant-gas security model.* We formalize a marker-aware write oracle M^+ that observes, for each persistent write, whether the update is Set, Clear, Same, or Update. We define constant-gas security as noninterference over traces that include ordinary constant-time observations and M^+ update classes. The definition includes explicit public declassification for returns, status, and public state, and separates declassified returns from ciphertext-only client returns.
- *Concrete evidence on a real confidential-EVM stack.* On stock Sapphire localnet, `/proc/<runtime-pid>/io:wchar` separates confidential executions that write logical zero from those that write logical one perfectly, i.e., a process-level counter is enough to reveal a secret bit encoded by a storage transition. For example, a confidential sealed-bid contract that records a private success flag can reveal whether a user crossed a zero/nonzero balance threshold, enabling deanonymization or targeted transaction strategies while the public gas receipt still appears padded.
- *A constant-gas programming system.* We present Coga, which combines manually constructed constant-gas primitives and a proof-carrying toolchain. The system uses constant-gas conditional select, marker-free scalar cells, fixed-schedule arrays and maps, control-flow linearization, local and private-position-map Path ORAM routes for larger secret-indexed state, and a restricted Solidity-like source language such as public environment reads, bounded arrays and mapping, single-value returns, and `require` guards with rollback under declared status policies.
- *An evaluation with end-to-end contracts and costs.* We evaluate the cost of each constant-gas primitive via microbenchmark and also the end-to-end Coga toolchain performance via realistic Solidity contracts. The results show that constant-gas scalar operations are reasonable, small maps are viable under public capacity bounds, and large maps remain expensive. For full contracts, especially those involving ERC20-style global maps, current constant-gas techniques are best understood as a proof-of-concept, not as an economically viable drop-in replacement.

2 Background and Threat Model

2.1 Confidential EVMs

Public blockchains deliberately reveal transaction contents and state updates. Confidential smart-contract platforms change this assumption. A user submits encrypted transaction data; the confidential runtime decrypts it inside a TEE; the contract executes over plaintext inside the enclave; and the resulting state remains encrypted outside the enclave. Oasis Sapphire is a concrete example of this design family: it is an EVM-compatible confidential ParaTime with encrypted transactions and confidential contract state [13, 14].

This design is attractive because it preserves Solidity, EVM-compatible tooling, and near-native execution speed. It, however, also inherits the side-channel problem of TEE-based confidential computing. TEE isolation protects plaintext memory from direct inspection, but it does not by itself guarantee that control flow, addresses, I/O volume, resource accounting, or storage-update metadata are independent of secrets [6, 10, 22]. Indeed, Sapphire's own developer guidance warns that taking actions based on private data may leak through time, gas use, and accessed memory locations, and recommends equalizing private code paths when needed [13]—i.e., the constant-time programming principle.

An important note for this paper is that "public gas is padded" and "the host learns nothing" are different claims. Sapphire provides `Sapphire.padGas` to equalize EVM execution gas to a pre-defined target, but the documentation explicitly scopes this mechanism to execution gas rather than all transaction-size or host-visible effects [12, 15]. Our model therefore distinguishes the public gas receipt from the stronger host-visible trace available to an untrusted, curious, or even malicious operator of the confidential runtime.

2.2 Storage layout and gas-sensitive updates

EVM storage is persistent key-value storage indexed by 256-bit slots. Smart contracts written in Solidity lay out state variables into slots. Specifically, for mappings, the storage location for a key is derived by hashing the key together with the mapping's base slot [19]. A conventional mapping is therefore conceptually unbounded and virtually initialized to zero for every key. This is convenient for programming, but it is dangerous for confidentiality: zero is a distinguished marker value in the storage model.

EIP-2200 gives structured definitions for net gas metering of storage writes, and later changes introduce cold/warm access costs and refund changes [4, 5, 21]. Table 1 summarizes these EIPs and gives the refund-adjusted net cost for writing to a global slot whose current value still equals its transaction-start value. While the concrete costing numbers are less important, they show that the execution stack has reason to treat storage updates differently depending on the old value, the new value, and whether zero is involved.

Table 1: Refund-adjusted `SSTORE` costs for clean slots under the latest EVM gas schedule. Negative cost means net refund.

Label	Update			Cold	Warm
Set	zero	→	nonzero	22,100	20,000
Clear	nonzero	→	zero	200	−1,900
Update	nonzero	→	nonzero	5,000	2,900
Same	nonzero	→	nonzero	2,200	100
Same	zero	→	zero	2,200	100

2.3 Threat model

We target confidential EVM deployments in which contract code is public, while selected calldata fields and persistent state values are confidential. The adversary may operate or compromise the host that runs the confidential runtime. The adversary cannot directly read enclave registers, plaintext memory, plaintext calldata, or plaintext storage values—the TEE hardware enclave is trusted in providing isolated and confidential computing. However, the adversary can observe public transaction metadata (including the entrypoint called in the contract), public outputs, host-visible I/O

and storage activity, and most importantly, **side-channel proxies for the runtime's storage-update behavior**. For encrypted client returns, the adversary also sees the ciphertext, nonce/AAD evidence, return shape, and session binding, but not the plaintext return unless it is separately declassified.

We assume that secrets may influence arithmetic values, but a safe contract must prevent secrets from influencing the following unless explicitly declassified: control flow, logical address schedules, write counts, write phases, update classes, public status, and public returns. The model deliberately covers a stronger host than an ordinary blockchain user. A public user may only see the receipt gas and public chain state. A malicious host may additionally see runtime-level effects that never appear in the receipt.

Freshness and envelope assumptions. For readability, we assume that encrypted storage records are protected with freshness or another mechanism that prevents ciphertext equality from becoming an additional channel. In the Coga implementation, the storage object layer supplies sealed records with fresh nonces, and the production acceptance path checks freshness and rollback counters. A target chain that bypasses these runtime objects must provide an equivalent property. We also assume that transaction envelope size is public; when necessary, a caller or wrapper must pad encrypted calldata separately from execution gas.

Out of scope. We do not claim protection against power analysis, electromagnetic leakage, all microarchitectural attacks, wall-clock scheduling channels, or compiler miscompilation. Solidity-level constant-gas code does *not* make arbitrary EVM-compatible clients such as Geth, Reth, or Besu side-channel safe. Those clients contain caches, tries, JIT or interpreter choices, state backends, and database paths that are outside the Solidity/EVM semantics.

3 The M^+ Constant-Gas Property

3.1 Marker-aware write observations

Let a program execution produce writes to logical locations. For a write to location ℓ, let *old* be the value stored immediately before the write and *new* the value stored immediately after it. Let D be a distinguished marker set; in EVM storage, the default marker is $D = \{0\}$. The marker-aware classification function is:

$$\mathrm{class}_D(old, new) = \begin{cases} \mathsf{SAME} & \text{if } old = new, \\ \mathsf{SET} & \text{if } old \in D \wedge new \notin D, \\ \mathsf{CLEAR} & \text{if } old \notin D \wedge new \in D, \\ \mathsf{UPDATE} & \text{if } old \notin D \wedge new \notin D \wedge old \neq new. \end{cases}$$

The ordinary constant-time memory observer sees the sequence of accessed locations. A transition observer also sees whether a write changed the value. The M^+ observer sees the sequence

$$\mathrm{Obs}_{M^+}(P, x) = \langle (\ell_i, \mathrm{class}_D(old_i, new_i)) \rangle_i,$$

possibly interleaved with the instruction trace, logical read/write trace, public gas schedule, and public outputs. This is the smart-contract form of update-obliviousness. The term M^+ emphasizes that the observer is stronger than the usual control-flow and memory address observer M behind the constant-time principle.

3.2 Security definition

Let an input x consist of public fields $pub(x)$ and secret fields $sec(x)$. Let $rel(P, x)$ be the declared release policy for the program, including public returns, public status, and public state updates that intentionally reveal secret-derived facts. For `encrypted_to_client` returns, the release policy includes the public ciphertext evidence and session binding, not the plaintext return.

Definition. A confidential entrypoint is *constant-gas secure* under M^+ if, for every pair of executions with equal public inputs, equal public environment, and equal initial public state, the observations outside the declared release policy are equal:

$$pub(x_1) = pub(x_2) \implies \mathrm{Obs}_{M^+}(P, x_1) \setminus rel(P, x_1) = \mathrm{Obs}_{M^+}(P, x_2) \setminus rel(P, x_2).$$

The definition is intentionally analogous to constant-time noninterference. What changes is the observation function. A program must not only execute the same public schedule and touch the same public locations; it must also induce the same public sequence of update classes. When ORAM-backed state is used, the equality of storage observations is replaced by the ORAM security relation under the stated cryptographic assumptions.

Strictness over constant-time. The security notion behind constant-gas programming is strictly stronger than constant-time programming. For example, consider a fixed slot containing zero and a branchless computation that stores $f(secret) \in \{0, 1\}$ into that slot. The instruction trace and slot trace are identical for all secrets. Yet M^+ sees Same for $0 \to 0$ and Set for $0 \to 1$. Even a weaker change-only observer is not enough to model the EVM case: with fixed old value 1 and new value either 0 or 2, every run changes the value, but M^+ distinguishes Clear from Update.

Why total gas alone is not the definition. A contract that happens to consume the same total receipt gas is not necessarily constant-gas secure. Total gas can be padded after the leak has already influenced storage traffic, backend metadata, or host-visible I/O volume. Conversely, a confidential runtime might pre-charge public gas before execution, which would make receipt gas useless as an attack channel while leaving the host-level M^+ channel intact. The property is therefore stated over traces, not merely over one public scalar.

4 Evidence from Oasis Sapphire

The M^+ model in §3 assumes that a host can observe more than the public gas. In this section, we show that this assumption is not impractical by simulating an untrusted host running a node for the Oasis Sapphire Localnet. The purpose of the experiment is to validate that whether a host-visible trace distinguish a confidential $0 \to 0$ storage transition from a confidential $0 \to 1$ storage transition when the bytecode, control flow, storage-slot sequence, write count, wrapped-calldata gas, and receipt gas are controlled.

4.1 Sapphire execution and the host observer

Sapphire is an EVM-compatible confidential ParaTime. A client can submit an encrypted transaction envelope; the confidential runtime decrypts it inside the execution boundary; and the contract can read and update confidential state without revealing the plaintext calldata or state value to ordinary chain observers [13, 14]. This

design changes what a public blockchain user sees, but it does not remove the host from the execution path. The host or node operator can still observe the public transaction envelope, process-level accounting, container counters, RPC timing, and other system effects of running the confidential runtime.

This distinction matters for gas padding. Sapphire exposes a gas-padding function, `Sapphire.padGas`. A contract can use this function to pad EVM execution gas to a chosen target [12, 15]. Padding is useful, but it is not the same as saying that every host-visible effect of the confidential execution has become independent of secrets. A wrapped call is still carried by a public outer transaction, and host-side process counters are outside the contract receipt. Thus our experiment treats receipt gas as a control, not as the only leakage target.

4.2 Workload and controls

The primary workload is `WrappedIdenticalBytecodeP1`, shown in full in Appendix A. It accepts a confidential `bytes32` token, classifies the token to a logical bit, writes that bit to a fixed contiguous range of storage slots, and then calls `Sapphire.padGas`. The experiment used `slotCount = 448` and `padTo = 10050000`. Each measured sample used a fresh contract deployment, so every tested slot was initially the EVM marker value zero.

```
function measurePadded(bytes32 token) external
{
  uint256 value = _classify(token);
  _writeAll(value);       // fixed slot range
  _padGas(padTo);         // fixed execution-gas target
}

function _classify(bytes32 token) internal pure
  returns (uint256 value)
{
  assembly ("memory-safe") {
    let a := iszero(xor(token, 0x2222...2222))
    let b := iszero(xor(token, 0x3333...3333))
    value := or(a, b)
  }
}
```

Figure 2: Simplified Sapphire workload. The classifier is branchless; `_writeAll` performs the same public loop over the same storage slots for every token.

The main pair compares two confidential tokens. The token `0x11...11` maps to logical zero, so the fresh slots receive $0 \to 0$ writes. The token `0x22...22` maps to logical one, so the same fresh slots receive $0 \to 1$ writes. A same-same control compares `0x22...22` with `0x33...33`; both map to logical one and therefore induce the same $0 \to 1$ update class. This control is important: it checks whether the measured signal is merely token-specific rather than update-class-specific.

The harness also controls the public envelope. Sapphire wrapped calls are encrypted, but the outer transaction data still has a public calldata-gas cost. For each host session, the harness calibrated candidate `publicTxDataGas` values across all token classes, selected a common target, and ensure every prepared and observed transaction matched that target exactly. In the experiment, all 216 measured wrapped calls had `publicTxDataGas = 2368`, all encrypted-call checks succeeded, and the runtime code hash was constant.

4.3 Results and interpretation

Table 2 gives the key experiment results. The channel is the runtime process's `io:wchar` field in `/proc/<pid>/io`, a Linux counter for bytes submitted to write-like I/O operations. Over the baseline main-pair samples, every $0 \to 1$ execution produced a larger `wchar` delta than every $0 \to 0$ execution. The public controls, as expected, do not show the separation: receipt gas, public wrapped-calldata gas, and adjusted gas all remain at chance-level discrimination. The same-same control also remains near chance, which is the expected outcome when both confidential tokens induce the same zero-to-nonzero update class.

Table 2: Primary Sapphire Localnet result. Public gas controls stay near chance (i.e., oblivious to confidential memory update class), but a host-visible runtime I/O counter separates them cleanly.

Key item	Result
Validated channel	runtime `io:wchar`
Main-pair AUC	1.000
Same-same control AUC	0.4175
`receiptGasUsed` AUC	0.4931
`publicTxDataGas` AUC	0.5000
`adjustedGas` AUC	0.4931
Replication	3/3 host sessions
$0 \to 0$ `io:wchar` range	4,444–8,134 bytes
$0 \to 1$ `io:wchar` range	74,671–77,317 bytes
Minimum class gap	66,537 bytes

We also performed a secondary broadening campaign. It replaces the equality-based classifier with the simpler branchless projection `value := and(token, 1)`, while keeping the same fixed-slot, fixed-control-flow, wrapped-call, and gas-padding structure. In that workload, the runtime `wchar` channel validates again with main AUC 1.000 and control AUC 0.4405; the compute-process `wchar` channel also validates with main AUC 1.000 and control AUC 0.5039; and two runtime scheduler counters validate with main AUC 0.9874 and control AUCs below 0.47. All four secondary channels reproduce in 3/3 sessions, and the public gas/data AUCs are 0.500. This secondary run shows that the primary result is not an artifact of the particular equality comparison used in Figure 2.

The security implication is immediate. A host observer does not need to decrypt calldata or state if the secret is encoded by a persistent-state transition. A contract that records whether a private predicate is false or true by leaving an empty slot empty or by setting it to a nonzero value can give the host an oracle for that predicate. For confidential smart contract execution, that predicate may be the fact that an authentication token matched, a sealed bid crossed a reserve, a hidden order crossed a price band, a balance moved from empty to nonempty, or a private risk check changed from fail to pass. In all of these examples, the leaked bit is not some incidental metadata, instead, it is the application secret that the confidential contract was intended to protect.

The alarming part of the result is its ordinariness. The validated primary channel is not a specialized microarchitectural probe and not a decrypted contract output. It is a host-visible process counter available at the layer that confidential execution already treats as untrusted. Moreover, the signal remains after the usual developer-level

mitigations used in this workload: branchless computation, fixed storage slots, fixed write count, confidential wrapping, exact public calldata-gas matching, and final gas padding. Thus the experiment instantiates the M^+ observer from §3.1 on a real Sapphire-style software stack. It also explains why constant-gas programming must reason about the semantic class of writes rather than only about public receipt gas.

The measurement was performed on stock Sapphire Localnet on a controlled Ubuntu host with SGX hardware. Due to ethical concerns, we did not experiment on the Sapphire Mainnet nor do we claim evidence that a production Sapphire deployment exposes exactly the same numeric channel. The result is instead mechanism evidence: on the unmodified Localnet stack used by Sapphire developers, confidential storage-update classes can be converted into host-visible I/O signals even when public gas observables are controlled. We hope this real-world evidence is sufficient to motivate the constant-gas programming notion argued in this paper.

5 Update-aware Leakage Patterns

The previous sections show how the update-class of a single SSTORE can leak secrets. That leak is one counterexample to constant-time programming, but it is not the only way in which Solidity code can make a secret visible to an M^+-style host. In an EVM execution, the persistent write classes sit inside a larger public schedule: which slots are touched, which slots are cold or warm, how much transient memory is allocated, whether logs or calls are emitted, and whether the call commits or aborts. A robust and practical constant-gas programming discipline for Solidity smart contracts must therefore protect all these identified sources of leaks.

Table 3 summarizes the leak patterns that determine the constant-gas programming principle presented in the rest of the paper. The table summarizes the Solidity/EVM surface that a source-based toolchain must either control, expose through an explicit release policy, or reject explicitly.

Secret-dependent locations and warmness. A Solidity mapping access is compiled to an EVM storage slot obtained by hashing the key with the mapping base slot. Hashing does not make the access schedule public: a secret key still selects a secret storage slot. However, the EVM distinguishes the first access to a slot in a transaction from later accesses to the same slot. Thus even fixed code with two reads can leak an equality predicate through the cold/warm pattern, as in `addressWarmth` in Figure 3. The constant-gas response is to make the logical address schedule public. For small maps this means iterating with a known capacity. For larger maps it means using an ORAM backend and then applying the same update-class discipline to ORAM buckets and metadata.

Variable working sets. EVM memory is transient rather than persistent, but it is still metered by the highest memory word reached during execution. Dynamic arrays, `string`, dynamic `bytes`, and variable-size ABI encodings can therefore turn a secret length into a resource trace. A hash operation such as `keccak256` is value-oblivious only when its input shape is fixed; its input length and memory range are part of the public schedule. The `memoryShape` example in Figure 3 leaks before any persistent storage update is considered. This is why in Coga, the accepted Solidity subset uses

```
contract RejectedPatterns {
  mapping(bytes32 => uint256) records;
  uint256 public count;
  uint256 private flag;
  bytes32 private scratch;
  event Hit(uint256 bit);

  function addressWarmth(bytes32 k0, bytes32 k1, uint256 secret)
    public
  {
    uint256 a = records[k0];
    bytes32 k = ((secret & 1) == 1) ? k0 : k1;
    uint256 b = records[k];
    // second touch is warm iff k == k0
    flag = a + b;
  }

  function memoryShape(uint256 secret)
    public
  {
    bytes memory b = new bytes(secret & 31);
    scratch = keccak256(b);
    // memory and hash length follow secret
  }

  function outputShape(uint256 secret)
    public returns (uint256)
  {
    if ((secret & 1) == 1) emit Hit(1);
    return secret & 1;
    // public unless explicitly declassified
  }

  function externalBoundary(address target, uint256 secret)
    public
  {
    if ((secret & 1) == 1) {
      (bool ok,) = target.call(abi.encodePacked(secret));
      require(ok);
      // call boundary and callee behavior are public
    }
  }

  function abortOrMetadata(bytes32 k, uint256 secret)
    public
  {
    require(secret != 0);
    // public status and rollback boundary

    if (records[k] == 0) count++;
    // occupancy metadata leaks insert vs update

    records[k] = secret;
  }
}
```

Figure 3: Representative Solidity patterns that are not admitted as confidential source constructs. They illustrate schedule leaks around storage updates rather than new attacks on the SSTORE value class itself.

fixed-size scalars, fixed arrays and structs, public loop bounds, and fixed-shape `keccak256` calls over fixed-size ABI inputs.

Effect count and phase. A branch may be linearized at the instruction level and still omit an effect. For example, a secret-dependent early return may skip the writes that would otherwise initialize, clear, or refresh a state object. A `break` in a scan loop similarly reveals the first matching index by the number of reads and writes that precede it. Under M^+, sequence length and write position are observations just as much as individual write classes. The source discipline therefore forbids `while`, `do while`, `break`, `continue`,

Table 3: Update-aware leakage patterns relevant to constant-gas programming. Each row is either a direct M^+ write-class source or a way to make the write-class schedule, host-visible work, or public outcome depend on a secret.

Pattern for leakage	Secret-dependent observation	Required discipline for mitigation
Raw persistent value class	zero-to-nonzero, nonzero-to-zero, same-value, and nonzero update classes	marker-free cells, fresh or rerandomized records, public write budgets
Secret location or warmness	slot address, first-touch/cold state, or aliasing between two touches	public access schedules; bounded full scans or ORAM for secret-indexed state
Variable working set	maximum memory offset, dynamic ABI size, dynamic-array length, or hash input length	fixed-size values, fixed arrays/structs, public loop bounds, fixed-shape ABI/hashing
Secret effect count or phase	number of reads/writes, skipped cleanup, loop exit point, early return, or commit point	control-flow linearization, predicated effects, no secret early exits
Public outputs and logs	event count, event topics/data size, return shape, or plaintext return value	no events in confidential entrypoints; explicit return/declassification policy
External boundary effects	callee address, call value/data/gas, success bit, callback schedule, or callee writes	external calls and callbacks rejected in the admitted confidential subset
Abort and metadata writes	public status, rollback, occupancy bits, versions, free lists, refund or cleanup records	status must be public/declassified; metadata follows public marker-free schedules

and early confidential returns, and rewrites secret branches into predicated effects over a fixed public order of instructions.

Public outputs and logs. Events are designed to be observed. Their topics, data length, and count appear in the public receipt/log stream, and their emission point also creates a public phase marker for the host. A return value has the same problem unless it is intentionally public or sent through a client-encrypted return channel. The `outputShape` example in Figure 3 therefore has two separate release questions: whether the log may be emitted, and whether the return bit may be public. In response, Coga explicitly require and enforce that confidential entrypoints do not emit events, and every entrypoint declares a return policy: forbidden, public, declassified, or encrypted to the client.

External boundaries. An external call exposes more than a local branch. The callee address, call data size, call value, forwarded gas, success status, reentrant callback schedule, and callee-side storage behavior can all vary with the secret. Even if the caller pads its own gas after the call, the host has already observed the boundary and the callee may have performed unmodeled writes. For this reason, Coga rejects external calls, callbacks, `try/catch`, fallback and receive functions, and inheritance or modifiers that could hide such calls in the current confidential subset.

Abort, rollback, and metadata. A `require` guard is not just a control-flow test. It also determines public success or failure, selects whether later writes occur, and decides whether earlier tentative writes commit or roll back. When the predicate is secret, this is a release of the predicate unless the status channel explicitly allows it. Similarly, state-object metadata such as occupancy bits, version counters, free lists, refund records, freshness counters, and cleanup markers can leak the same information as the value path if they use raw zero markers or secret-dependent schedules. The `abortOrMetadata` example in Figure 3 leaks both through status and through the `count` update. The constant-gas response is to treat metadata as confidential state: it must be marker-free, updated on a public schedule, and covered by the same rollback and status policy as the logical value.

Summary. These patterns explain the shape of the design that follows. A confidential contract may compute secret values, but secrets must not choose addresses, object capacities, loop trip counts, output shapes, boundary calls, abort status, or metadata schedules unless the choice is part of the declared release policy. Everything else is lowered to public schedules and marker-free storage objects, or the program is rejected. The next two sections follow this summary directly: §6 states the programming discipline, including the source boundary, and §7 explains how Coga enforces that boundary.

6 Constant-Gas Programming

The leakage patterns in §5 confines design space. A confidential contract may compute on secrets, but the execution must not let those secrets choose the public schedule, the shape of an output, or the semantic class of a persistent write. Constant-gas programming is the source-based discipline that enforces this separation. It has two halves. The source language fixes the schedules that the hostile host can count: instructions, loops, object accesses, returns, status, and boundary events. The storage-object layer then realizes each scheduled state operation using representations whose physical update class does not depend on the logical secret value.

This split is important. A branchless conditional select does not, by itself, remove marker writes. It only turns a secret control decision into a data value. Likewise, a full-scan map or an ORAM backend does not, by itself, remove marker writes. It fixes or hides the address schedule. The marker leak disappears only when the scheduled object writes are also encoded so that logical zero, empty entries, unchanged values, and ordinary updates induce the same public write-class schedule. Coga therefore treats the program, the sidecar policy, and the state-object backend as one admitted package. If any part of the package leaves a secret-dependent schedule or release unaccounted for, the package is rejected.

Table 4: Source-language boundary for proof-grade confidential entrypoints. Each row groups an admitted public-shape construct with the corresponding constructs that remain outside the current subset.

Category	Admitted public shape	Rejected or out of scope
Values	`bool`, `uint256`, `bytes32`, `address`; fixed arrays and structs	Dynamic arrays, `string`, dynamic `bytes`, other integer widths
Expressions	Arithmetic, bitwise, comparison, and boolean expressions with supported assignment targets	Inline assembly; unsupported assignment targets
Control flow	Predicated `if`/`else`; public-bounded `for` loops	`while`, `do while`, `break`, `continue`; early confidential returns
Helpers and context	Internal pure/view helpers with one return expression; public reads of `msg.sender`, `tx.origin`, and selected `block.*` fields	Multi-statement helpers; external calls, callbacks, inheritance, modifiers, constructors, fallback/receive functions
Failures and effects	`require(expr)` guards governed by an explicit status policy	Arbitrary `revert`, custom errors, `try/catch`; event emission
Hashing and ABI	Fixed-shape `keccak256(abi.encodePacked(...))` over fixed-size values	Dynamic ABI shapes
Confidential storage	Cells, fixed arrays, and fixed-capacity maps	Unbounded mappings; raw confidential storage backends
Release channels	Single return values marked public, declassified, forbidden, or `encrypted_to_client`	Multiple return values; writing secret data into public state

6.1 From leakage patterns to a source boundary

Only a subset of Solidity language features, summarized in Table 4, is allowed in constant-gas programming. This is not a convenience restriction for the compiler. Instead, it is implied by §5. Each admitted construct has a public-shape interpretation under M^+; each rejected construct corresponds to a row of Table 3 whose schedule or outcome would otherwise have to be modeled as public. For example, a fixed array has a public length, but a dynamic array has a secret-usable length and memory footprint. A bounded `for` loop has a public trip count, but `while`, `break`, and `continue` can make the number or phase of effects secret-dependent. A mapping is admitted only after the sidecar gives it a public capacity and a constant-gas backend; a raw Solidity mapping remains a virtual zero-initialized address space and is not a confidential object.

The source boundary also fixes what counts as an intentional release. Public environment reads are modeled as public inputs, not as hidden state. A public return, a declassified return, a declassified success bit, and an encrypted-to-client return are different policies. The first two expose plaintext according to the sidecar; the encrypted-to-client policy exposes only ciphertext evidence, canonical length, nonce/AAD evidence, and session binding to the host. Events are not admitted for confidential entrypoints because their count, topics, data length, and emission point are public by construction. External calls are rejected for the same reason: the callee, call data, forwarded gas, success bit, callback schedule, and callee writes would become part of a public boundary not covered by the local object proof.

6.2 Predicated effects and constant-gas select

The control-flow rule follows from the "*effect count and phase*" row of Table 3. A secret-dependent branch must not decide whether a storage operation happens. Instead, both sides are executed in a public order, and every assignment that would have occurred on one side becomes a predicated assignment. This means replacing statement like `if (p) { x = e }` with `x = select(p, e, x)`. When `e` is stored in a global slot, the predicate does not have an effect on the global operation: the slot is always read, but whether the read value will be assign to `x` is guarded by the predicate.

Figure 4 shows the manually constructed constant-gas select primitive for a fixed-width scalar. Later the Coga compiler uses the same semantic operator in its intermediate language. The constant-gas version uses the predicate only to select data, not to select which effects occur because this primitive requires loading values from both sides. When one or both sides involves storage operations, this lets Coga to prove that the same scheduled storage operations are executed regardless of whether the predicate is true or false.

```
function cg_select_u256(bool c, uint256 t, uint256 f)
  public pure returns (uint256 r)
{
  uint8 v; uint256 m;
  assembly {
    v := iszero(c)
    v := shl(7, v)
    m := signextend(0, v)
    m := sar(8, m)
    r := xor(t, f)
    r := and(m, r)
    r := xor(r, t)
  }
}
```

Figure 4: A constant-gas select primitive for `uint256`. The toolchain uses this operation at the expression level; the storage-object layer is responsible for making the resulting physical writes marker-free.

6.3 Marker-free cells and arrays

The select primitive in §6.2 fixes how a secret predicate chooses a value. It does not fix how the chosen value is represented in persistent storage. When `x` is stored globally, if the selected value is assigned to a raw Solidity storage variable, the final `SSTORE` still exposes whether the physical slot moved through zero, stayed unchanged, or changed from one nonzero word to another. Thus a

scalar storage assignment needs its own constant-gas object operation, not merely a branchless expression on the right-hand side.

Figure 5 shows a representative construction for storing scalar in a constant-gas way. The scalar code contrasts a raw `uint256` slot with a direct-Solidity marker-free cell. The direct cell stores the logical value at a fresh public `version`. Logical zero is encoded as the nonzero payload `MARK_ZERO` plus a nonzero status word whose zero bit is set; logical nonzero values store the payload itself plus the base status word. Every `set` therefore writes the public `version` slot and two fresh nonzero record slots. The value being written can be zero, nonzero, or equal to the old logical value, but the physical record writes follow the same public update-class schedule. The production object layer may realize the same invariant by rerandomizing a fixed sealed record rather than by appending versions; the point is that logical zero and logical equality are not represented by the physical marker cases visible to M^+.

```
contract VerbatimU256 {
  uint256 private v;

  function get() public view returns (uint256) {
    return v;
  }

  function set(uint256 value) public {
    v = value;
  }
}

contract CGU256 {
  uint256 constant MARK_ZERO = 0xffff...ffff;
  uint8 constant STATUS_BASE = 1;
  uint8 constant STATUS_BIT_IS_ZERO = 1;

  uint256 private version;
  mapping(uint256 => uint256) private record;
  mapping(uint256 => uint8) private status;

  constructor(uint256 value) {
    set(value);
  }

  function get() public view returns (uint256) {
    bool z = u8_is_bit_set(status[version], STATUS_BIT_IS_ZERO);
    return cg_select_u256(z, 0, record[version]);
  }

  function set(uint256 value) public {
    bool z = (value == 0);
    version += 1;
    record[version] = cg_select_u256(z, MARK_ZERO, value);
    status[version] = cg_select_u8(z,
      u8_set_bit(STATUS_BASE, STATUS_BIT_IS_ZERO), STATUS_BASE);
  }
}
```

Figure 5: Marker-free scalar cell. The constant-gas version contrasts a raw scalar slot with a representation that avoids marker-aware zero and same-value updates.

A fixed array is the same object discipline lifted over a public list of cells, as demonstrated in Figure 6. The array length is public, and a secret index is used only to compute data predicates inside a scan. In `CGArray64U256.get`, every cell is read in index order and the return accumulator is updated by `cg_select_u256` when the public loop index equals the secret logical index. In `CGArray64U256.set`, every cell is also committed in index order. The selected cell receives the new logical value; all other cells receive their old logical values through the same marker-free `CGU256.set` operation. Therefore an inactive predicate, an out-of-range secret index under the declared policy, a write of zero, and a write of the previous value all keep the same physical write boundary. This is more expensive than a raw array access, but it is the storage analogue of constant-time scanning under an observer that also sees update classes.

```
contract VerbatimArray64U256 {
  uint256[64] private values;

  function get(uint256 index) public view returns (uint256) {
    return values[index];
  }

  function set(uint256 index, uint256 value) public {
    values[index] = value;
  }
}

contract CGArray64 {
  uint256 constant N = 64;
  CGU256[N] private cells;

  constructor(uint256[64] memory values) {
    for (uint256 i = 0; i < N; i++) {
      cells[i] = new CGU256(values[i]);
    }
  }

  function get(uint256 index) public view returns (uint256 r) {
    r = 0;
    for (uint256 i = 0; i < N; i++) {
      uint256 x = cells[i].get();                    // CGU256.get()
      r = cg_select_u256(i == index, x, r);
    }
  }

  function set(uint256 index, uint256 value, bool p) public {
    for (uint256 i = 0; i < N; i++) {
      uint256 old = cells[i].get();                  // CGU256.get()
      bool hit = p && (i == index);
      cells[i].set(cg_select_u256(hit, value, old));  // CGU256.set()
    }
  }
}
```

Figure 6: Marker-free fixed array. The constant-gas version contrasts a raw array with an array representation whose secret predicates are consumed by select and by object operations, not by branches that change the storage schedule.

6.4 Bounded maps

Mappings are the most challenging obstacle for constant-gas Solidity because a raw mapping has no public length, no iterator, and a virtual zero value for every key. The verbatim code in Figure 7 therefore has two independent problems when `key` or `value` is confidential: `table[key]` selects a secret hashed storage slot, and `table[key] = value` exposes the marker class of that slot. A constant-gas bounded map replaces the raw mapping by a state object with a public capacity and marker-free entries.

The constructor in Figure 7 is part of the public object setup. It allocates exactly `CAPACITY` entries and initializes each entry to the same marker-free representation of logical empty: logical key zero, logical value zero, and nonzero status metadata. The capacity is therefore not a secret parameter; it is a public part of the object type and of the sidecar policy. As with scalar cells, the direct-library version shown here uses a public `version` index so that later writes

```
contract VerbatimMapU256ToU256 {
  mapping(uint256 => uint256) private table;
  function get(uint256 key) public view returns (uint256) {
    return table[key];
  }
  function set(uint256 key, uint256 value) public {
    table[key] = value;
  }
}

contract CGMapU256ToU256 {
  uint256 constant MARK_ZERO = 0xffff...ffff;
  uint8 constant STATUS_BASE = 1;
  uint8 constant STATUS_BIT_K_IS_ZERO = 1;
  uint8 constant STATUS_BIT_V_IS_ZERO = 2;

  struct Entry {
    mapping(uint256 => uint256) key;
    mapping(uint256 => uint256) value;
    mapping(uint256 => uint8) status;
  }

  uint16 public immutable CAPACITY;
  Entry[] private entries;
  uint256 private version;

  constructor(uint16 capacity) {
    version = 1;
    for (uint16 i = 0; i < capacity; i++) {
      entries.push();
      entries[i].key[version] = MARK_ZERO;
      entries[i].value[version] = MARK_ZERO;
      entries[i].status[version] = u8_set_bit(
        u8_set_bit(STATUS_BASE, STATUS_BIT_K_IS_ZERO),
        STATUS_BIT_V_IS_ZERO);
    }
    CAPACITY = capacity;
  }
}

function get(uint256 key) public view returns (uint256 r) {
  r = 0;
  bool k_is0 = (key == 0);

  for (uint16 i = 0; i < CAPACITY; i++) {
    Entry storage e = entries[i];
    uint256 ek = e.key[version];
    uint256 ev = e.value[version];
    uint8 es = e.status[version];

    bool ek_is0 = u8_is_bit_set(es, STATUS_BIT_K_IS_ZERO);
    bool ev_is0 = u8_is_bit_set(es, STATUS_BIT_V_IS_ZERO);
    uint256 ek_out = cg_select_u256(ek_is0, 0, ek);
    uint256 ev_out = cg_select_u256(ev_is0, 0, ev);

    bool hit_zero = k_is0 && ek_is0;
    bool hit_nonzero = (!k_is0) && (!ek_is0) && (key == ek_out);
    bool hit = (!ev_is0) && (hit_zero || hit_nonzero);
    r = cg_select_u256(hit, ev_out, r);
  }
}
```

Figure 7: A bounded constant-gas map for `uint256 => uint256` and its `get` operator. Logical zero keys and values are represented by status bits and nonzero payload words; lookup predicates choose the returned value but not which entries are visited.

go to fresh initialized record positions. A sealed-record backend can instead rewrite fixed buckets, provided that the rewrite is fresh and marker-free.

The `get` operation in Figure 7 is the read-side counterpart of `array_get`. The loop always reads all entries in the same order. For each entry, status bits decode whether the stored key or value is logical zero, and the equality test produces a data predicate. The result accumulator is updated only through `cg_select_u256`. Thus a miss, a hit in the first entry, and a hit in the last entry have the same entry-read schedule. Whether the returned plaintext is public is not decided by the map; it is governed by the entrypoint's release policy.

The update operation uses the same ingredients but commits records. It is not lowered to `table[key] = value`. As shown in Appendix B, a bounded-map `set` scans all entries at the current public version, computes predicates such as "this nonempty entry matches the key" and "this is the first empty entry," and writes a next-version record for every entry. A logical overwrite selects the new encoded value for the matching entry; a logical deletion selects the empty encoding for that entry; an insertion selects the first empty entry when the input value is nonzero; and all nonselected entries are copied through the same marker-free commit. The success or overflow bit is therefore an ordinary status result: it must be public by construction or explicitly declassified, not accidentally leaked through a skipped write or a different marker class.

6.5 ORAM-backed maps

The full-scan map in §6.4 is exact but linear in the public capacity. For larger maps, Coga therefore supports Path ORAM-backed maps [20]. At the level needed for this paper, an ORAM is a randomized storage protocol that lets a private runtime access a logical block while exposing only a public-shape sequence of physical storage operations. Path ORAM stores encrypted blocks in the buckets of a binary tree. Each logical block is assigned to a random leaf. A block may reside either in the private stash or somewhere on the path from the root to that leaf. To access a key, the runtime privately determines the current leaf for the corresponding logical slot, reads every bucket on the root-to-leaf path, scans the stash and path contents with masked code, remaps the touched logical block to a fresh random leaf, and then writes back a fixed path plus fixed metadata. The public path length is determined only by the public capacity; the observed leaf sequence is protected by the Path ORAM theorem, the fresh-leaf distribution, the bucket and metadata encryption assumptions, and the stash-overflow bound.

In Coga, an ORAM-backed Solidity mapping is still a bounded state object. The compiler counts every lowered map `get` and `set` as a logical ORAM access after control-flow linearization and loop unrolling. The manifest then fixes the ORAM shape: leaf count, tree height, bucket count, block size, stash bound, metadata-page count, buckets per access, and the public commit-operation budget. A predicated write whose predicate is false still consumes the same ORAM access and refreshes the same public path and metadata schedule; only the logical value selected inside the private runtime differs. Thus ORAM is used for the address-schedule problem, not as a replacement for the update-oblivious object discipline. Internal states of the ORAM (e.g., encrypted buckets, filler blocks, stash records, etc) are all sealed or resealed through fixed-shape marker-free records, so logical miss, insert, update, delete, and no-op cases do not select different physical write boundaries.

The current prototype has two Path ORAM routes. The first route is the bounded *local-position-map* route. Here the Path ORAM position map—the private table mapping each logical slot to its current leaf—is part of the hidden runtime state for the object.

The runtime stores it as encrypted metadata pages, opens those pages inside the confidential boundary, scans them in a fixed public shape, and reseals the pages after each access. Because the proof-facing state contains the concrete position-map witnesses, fresh-leaf evidence, and before/after ORAM transition family for the accessed object, this route composes multiple accesses to the same map in one entrypoint: a read followed by a write simply checks the second access against the hidden state produced by the first. The cost is that the checked evidence grows with the local position map and with the number of accesses. The proof-grade implementation therefore caps this route at capacity 128. This cap is not a Path ORAM cryptographic limit; it is the largest local-position-map route that the current Rust manifest checker, Lean imported-shape checker, runtime witness extractor, and artifact-size budget admit without switching to a symbolic recursive position-map argument. Raising the constant alone would only make the implementation accept states for which the current theorem-facing route has not promised a bounded, reviewable witness family.

The second route is the *private-position-map* route for larger maps. It treats the position metadata itself as private theorem-facing state rather than as an enumerated local table in the main proof object. Concretely, the manifest records a one-level recursive position-map shape, and the proof artifact carries a compact symbolic certificate tying the object capacity, leaf count, recursion depth, and validation-step bound to that shape. This is the route used when a map is too large for the local cap: the certificate summarizes the large position-map family instead of materializing every leaf assignment and fresh-coin witness in the artifact.

The price of the private-position-map route is sequential composition. The current checked theorem covers one logical access to a given map in an entrypoint: one hidden before-state, one fresh remapping, one path rewrite, and one after-state related by the Path ORAM security interface. Two accesses to the same private-position-map object would require a stronger composition theorem that connects the first hidden after-state to the second hidden before-state, threads the fresh-coin plan through the recursive position-map access, and composes the stash-overflow bound and metadata-trace relation across the sequence. That multi-access theorem is not part of the present proof-grade route. Coga therefore rejects such entrypoints instead of silently assuming the composition step. In short, the local-position-map route is small enough to enumerate and therefore supports multiple accesses; the private-position-map route is scalable because it is symbolic, but is currently single-access per map per entrypoint.

6.6 Status, rollback, and returns

A Solidity `require` guard is a public event unless the system says otherwise. It determines whether the call succeeds, which writes commit, what a caller observes, and how later calls compose with the failed one. For this reason, Coga does not silently replace arbitrary reverts by a private flag. Arbitrary `revert`, custom errors, `try/catch`, and external calls are outside the current confidential subset. The allowed form is `require(expr)` under a status policy. If the guard is public-determined, the status may be public. If the guard depends on a secret, the sidecar (described in §7.1) must declassify the success bit explicitly.

On exact backends, a failed admitted guard rolls back transactional state while preserving the modeled public budget and the public abort outcome. An in-bound call commits through the same object discipline described above. For ORAM-backed state, the current proof-grade route is more conservative around mid-call aborts: edge cases that would require hidden ORAM state reconciliation are rejected rather than assigned an unproved rollback semantics. Return values follow the same principle. The source may have no return, a public or declassified single return, or an encrypted-to-client return. What is not allowed is for a secret result to leak again through a shorter schedule, a missing write, a cheaper update class, or a different public output shape.

7 The Coga Toolchain

While the primitives described in §6 enable manual development or inspection of constant-gas Solidity programs, such a practice can be tedious and error prone. A line such as `balances[k] = v` hides at least three security-relevant choices: whether `k` is secret, whether the target object has a public capacity and backend, and whether the eventual physical write is marker-free. Coga makes these choices explicit by compiling an admitted source package and policy sidecar into a proof-carrying artifact executed by a modeled runtime. It is not a bytecode compiler that makes arbitrary Solidity safe on a stock EVM client. It is a source-to-artifact toolchain for the restricted subset of Solidity described in §6.1 and leveraging the runtime/storage-object primitives described in §6.2–§6.6.

7.1 Package policy and sidecar metadata

The sidecar is required by Coga and contributes to the constant-gas guarantee (i.e., safety against the M^+ model). It names the confidential entrypoints, secret arguments, state objects, public capacities, backends, return policies, status policies, modeled public gas budgets, initializer requirements, and declassification labels. The compiler rejects missing state entries, unknown metadata fields, raw confidential backends, undeclared releases, plaintext declassification for encrypted returns, and confidential writes to public state. This is how the informal phrase "unless explicitly declassified" in §3.2 becomes a checked property enforced by Coga.

Figure 8 illustrates the role of the sidecar on a mixer-style contract. The source-level mapping `balances` is not accepted as an unbounded Solidity mapping. The sidecar turns it into a capacity-8 full-scan confidential map. The `withdraw` entrypoint may reveal whether the withdrawal succeeded, because that fact is part of the API, but the returned value is delivered through the encrypted client channel. The compiler and runtime must therefore prevent the same success bit from also changing the storage schedule, public gas, or write-class trace except through the declared status release.

7.2 Admission and lowering

Compilation begins by parsing the Solidity source and resolving it into our admitted subset (see §6.1). The front end rejects unsupported functions and statements before any security claim is made. It then checks that every externally callable function has a sidecar stanza, that secret arguments exist, that public environment reads are among the admitted fields, that confidential entrypoints target only confidential non-raw objects, and that the declared return

```
contract: ConfidentialMixer
entrypoints:
  deposit(bytes32,uint256):
    confidential: true
    secret_args: [commitment, amount]
    public_gas_budget: 16384
    returns: forbidden
    status: public
  withdraw(bytes32,uint256):
    confidential: true
    secret_args: [commitment, amount]
    public_gas_budget: 16384
    returns: encrypted_to_client
    status: declassified
    declassifies: [success]
state:
  balances:
    kind: map
    key: bytes32
    value: uint256
    capacity: 8
    backend: full_scan
    confidential: true
  total:
    kind: cell
    value: uint256
    backend: full_scan
    confidential: true
```

Figure 8: Example sidecar policy for a bounded confidential mixer. The capacity and backend are public design choices; the success status is an intentional release; the encrypted return exposes ciphertext evidence rather than plaintext.

and status policies match the function body. A secret-dependent `require` without `status: declassified` fails closed; so does an attempt to write a secret-derived value into public state.

For an admitted confidential entrypoint, the compiler lowers the body to a small update-oblivious IR. Public-bounded `for` loops are unrolled using compile-time bounds. Secret `if`/`else` statements are linearized by deriving a predicate for each block and appending both arms in a deterministic order. Local assignments are guarded by select. Cell, array, and map writes carry an explicit predicate into the object operation. A return under a nontrivial predicate is rejected, because it would make the return point itself a secret-dependent phase marker. The resulting artifact records a public gas schedule modeled after EVM bytecode gas, but this gas schedule can be changed by the deployer.

7.3 Control-flow linearization

Control-flow linearization (CFL) is the pass that turns a secret branch into an unconditional schedule of predicated effects. In ordinary code, the condition of an `if` statement decides which basic blocks execute. Under M^+, that decision is itself observable: it can change the instruction sequence, the number and order of object operations, the phase of a write, or the point at which a call aborts or returns. CFL changes what the condition is allowed to do. The condition may compute a boolean value that guards assignments, but it may not decide whether a scheduled effect is reached. This is the same high-level idea used by constant-time CFL systems, specialized here to the structured source boundary of Coga [3, 17, 18].

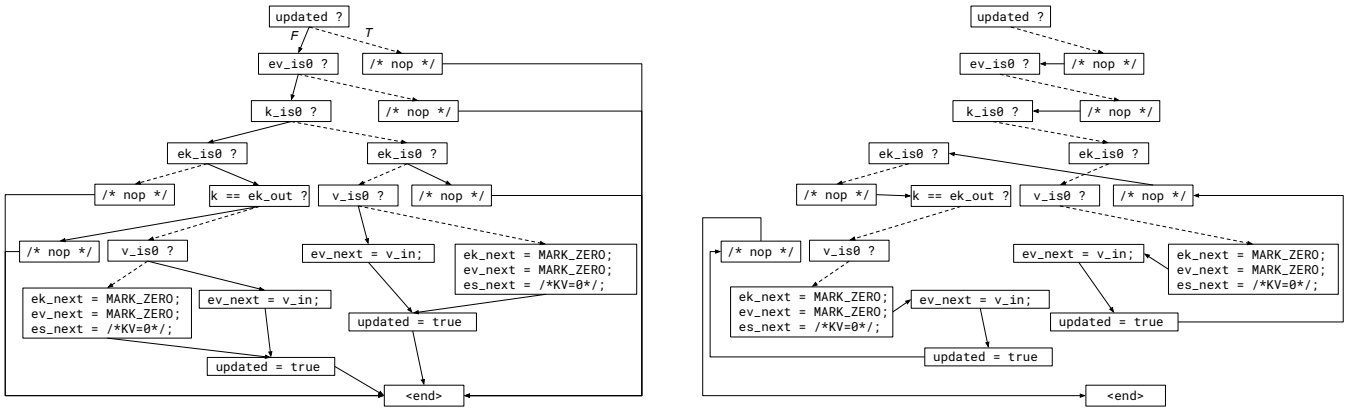
General CFL is often described as a CFG transformation: choose a deterministic order of basic blocks, compute for each block the predicate under which the original execution would have reached it, and then emit the blocks in that order with predicated effects. Coga does not need a general transformation over arbitrary EVM bytecode. Admission has already ruled out unstructured jumps, secret-dependent loop exits, and early confidential returns. The compiler therefore applies a syntax-directed version of CFL after lowering to the update-oblivious IR. It carries a path predicate, written here as pc, initially set to `true`. For a branch with condition c, the compiler emits the then-body under $pc \wedge c$ and the else-body under $pc \wedge \neg c$, using a fixed then-before-else order. Public-bounded `for` loops are unrolled first, so the same rule applies independently to every materialized iteration.

Leaf statements are then rewritten according to the kind of effect they have. A local assignment under pc becomes a select, for example `x = select(pc, e, x)`. A declaration under a nontrivial predicate receives its initializer when the predicate holds and the type-zero value otherwise. A storage-object write is not hidden inside a select. Instead, the compiler emits the same object operation and attaches the predicate to it: `cell_set(o, e, pc)`, `array_set(o, i, e, pc)`, or `map_set(o, k, e, pc)`, which then maps to the primitives described in §6.3 and §6.4 (or §6.5 if ORAM is requested). When the predicate is false, the runtime object operation still consumes the same public object schedule and preserves the old logical value. An admitted `require(c)` similarly becomes `abort_if(pc && !c)`: it sets the transaction abort flag when the guarded check fails, but the linearized suffix is still executed and rollback is handled at the commit boundary. A return is allowed only at the final top-level return point. A return under a nontrivial predicate is rejected because the return point would otherwise become a secret-dependent phase marker.

Select is therefore only a data operator in the lowered program. It is not a place to hide effects. Lowering accepts select arms only when they are pure branchless expressions. If one arm would perform a storage access, that access must appear as an explicit scheduled operation before the select, or the program is rejected. This restriction is what makes the statement schedule recorded in the artifact a public object: all storage reads, storage writes, aborts, commits, and returns appear at fixed positions in the linearized IR.

A source map assignment illustrates where the predicate finally lands. When source code writes `m[k] = v` under path predicate p, the compiler emits a single predicated `MapSet` statement. For a full-scan bounded map, the runtime object implementation executes a public pass of the form shown in Figure 10. The pass computes case predicates for update, insert, and delete cases, but every entry is read and later committed on the public schedule. Thus the predicate chooses the next logical entry, not which entry is visited or written.

Figure 9 is a CFG-level view of the same transformation illustrated in Appendix B in alternated code and comments. The left-hand graph is the natural nested implementation of one map-entry set operation (code in comments). The right-hand graph should not be read as a new secret-dependent CFG (code in foreground). It is the fixed order in which the linearized program evaluates block predicates and reaches effect boundaries. A block whose predicate is false still has the same leaf-statement shape: local assignments select the old value, and object writes carry a false predicate into the object operation.



(a) Branch structure from one map-update pass. **(b) Linearized public order with predicated effects.**

Figure 9: Control-flow linearization in Coga. The left graph shows the natural branch structure of one bounded-map case analysis. The right graph shows the deterministic order after CFL: branch tests produce predicates, and false predicates suppress only the logical update, not the scheduled effect boundary.

```
bool matched = false;
bool inserted = false;
for (uint256 i = 0; i < CAPACITY; i++) {
  Entry memory e = readEntry(i);       // public scan
  bool occ = occupied(e);
  bool hit = occ && keysEqual(e.key, k);

  bool del = p && hit && v_is0;
  bool ins = p && !v_is0 && !matched && !inserted && !occ;
  bool wr  = p && !v_is0 && (hit || ins);

  Entry memory out = e;
  out.key      = select(ins, k_in, e.key);
  out.value    = select(del, ZERO_VALUE,
                        select(wr, v_in, e.value));
  out.occupied = (occ && !del) || ins;

  matched  = matched  || hit;
  inserted = inserted || ins;
  writeEntry(i, out);                  // always scheduled
}
bool overflow = p && !v_is0 && !matched && !inserted;
```

Figure 10: One bounded-map update pass after predicates have been made explicit. Here p is the predicate attached to the source-level map write. The entry payloads, logical zero value, and occupancy metadata are encoded by the marker-free object layer; CFL fixes the scan and effect order.

After CFL, two executions with the same public inputs have the same linearized statement skeleton and the same positions for object operations. They may still differ in predicate values and in the data selected by those predicates. That remaining difference is precisely why CFL is paired with the storage-object layer. CFL removes the control and phase leak; marker-free cells, full-scan maps, and ORAM-backed maps remove the update-class and address-schedule leaks for the object operations that CFL has made public.

7.4 Runtime observer and proof artifacts

The runtime part of Coga executes artifacts compiled by Coga, not arbitrary Solidity bytecode. It supplies EVM-shaped values, entrypoint dispatch, public environment bindings, modeled resource accounting, return-channel handling, transactional rollback, and the persistent objects used by the exact and ORAM backends. Each run emits a transcript from which the hostile-host view is projected: public outcome, boundary schedule, logical object operations, physical reads and writes, write classes, modeled gas, status, and return evidence. For exact backends, the proof obligation is equality of this view outside the declared release policy. For ORAM-backed maps, the public components are equal and the storage component is related by the Path ORAM theorem route.

The compiler is not trusted merely because it produced the artifact. Each artifact carries a canonical admitted-package identity hash over the the executable-semantics profile, source bytes, and canonical sidecar metadata, That profile pins the semantics used by the proof: modular `uint256` arithmetic, fixed-size canonical ABI encodings, modeled `require`/abort rollback, the admitted public context variables, and the hash abstraction used by the proof-facing semantics. It is not a theorem about all Solidity 0.8 behavior.

The runtime bridge is checked by Lean-facing proof objects. The checked route includes source/artifact translation evidence, controller-observer certificates, hidden Path ORAM certificates, runtime kernel witnesses, trusted-kernel slot bundles, shell-boundary bundles, confidential-core bundles, refinement witnesses, seeded witness families, and canonical Path ORAM pair artifacts. For exact backends, the resulting theorem states that any two accepted executions of the same admitted package with equal public inputs, public environment, and initial public state have equal hostile-host observations outside the release policy. For ORAM backends, the theorem is conditional on the explicit cryptographic assumptions and stash-overflow bound associated with the chosen backend.

8 Evaluation

The evaluation has two parts. First, we report gas measurements and overhead on the constant-gas primitives (§6.2, §6.3, §6.4) which suggest where some of the constant-gas cost comes from. Second, we evaluate the proof-carrying toolchain (Coga compiler and VM)

on end-to-end contracts, including leakage controls, backend comparisons, abort/rollback edge cases, and proof-artifact size gates, because microbenchmarks alone do not answer whether the toolchain can admit complete stateful programs.

8.1 Primitive gas costs

The primitive experiments use Foundry and Solidity `0.8.31`. Each benchmark compares an efficient non-constant-gas implementation against a semantically equivalent constant-gas version. Gas constancy is checked by differential fuzzing, EVM trace hooks, and the total gas cost as a bulk check.

Conditional select. Being a fundamental operator, the overhead of the constant-gas conditional select plays an important role in the overall penalty of a constant-gas program. Table 5 shows the gas overhead of our implementation (§6.2) on primitive types with various lengths in bytes, compared with either a ternary operator or an if-else version, whichever is better. The constant-gas implementation incurs a reasonable overhead (less than 30%) which increases as the byte-length of the underlying type. The constant-gas version even outperforms the `if-else` implementation on the `uint8` type slightly in its worst case.

Table 5: Gas cost of constant-gas conditional select.

Type	Implementation	min	ave	max	const
uint256	ternary	830	837	845	
	if-else	940	947	955	
	Coga (overhead)	27.5%	26.4%	25.2%	1058
uint64	ternary	914	920	929	
	if-else	892	898	907	
	Coga (overhead)	18.6%	17.8%	16.6%	1058
uint8	ternary	826	833	841	
	if-else	936	943	951	
	Coga (overhead)	14.8%	13.8%	12.7%	948

Primitive type on storage. With a significantly heavier modeling of primitive types on storage (§6.3), we expect the overhead of the constant-gas versions to be high, as confirmed in Table 6. In general, the constant-gas versions cost approximately 5.4x more gas for a read access and around 1.0x - 2.6x more gas for a write access—a consistent trend across various types of primitives.

Mapping type on storage Storing a constant-gas version of a `mapping` on-chain can be expensive, as already hinted in §6.4, especially by the shear contrast of the implementation complexity shown in Figure 7. And the cost grows in linear form as the capacity of the `CGMap` grows, because every read or write to a `CGMap` in global storage requires one and two complete iterations of all pre-allocated entries, respectively. This trend is also confirmed in Table 7 across `mappings` of various key and value types. Even when the `CGMap` holds at most 256 (key, *non-zero* value) pairs, the `CGMap` might incur an 2300x gas overhead for read accesses and more than 1000x for write accesses on average.

The results are useful precisely because they are not flattering. Constant-gas select is cheap enough to use pervasively. Scalar storage roughly doubles write cost and increases read cost by about 5.4×.

Table 6: Gas cost for constant-gas scalar storage.

Type	Fun.	min	ave	max	const
`uint256`	`get`	2651	2651	2651	
`CGU256`	`get` (overhead)	544%	544%	544%	17065
`uint256`	`set`	24073	43786	44357	
`CGU256`	`set` (overhead)	262%	98.8%	96.3%	87062
`uint64`	`get`	2663	2663	2663	
`CGU64`	`get` (overhead)	538%	538%	538%	16982
`uint64`	`set`	24089	43328	44085	
`CGU64`	`set` (overhead)	258%	99.1%	95.7%	86253
`uint8`	`get`	2663	2663	2663	
`CGU8`	`get` (overhead)	540%	540%	540%	17048
`uint8`	`set`	24089	41589	44001	
`CGU8`	`set` (overhead)	257%	107%	95.4%	85974

Table 7: Gas cost for full-scan constant-gas maps. Costs are for mapping `u256 => u256` (modeled by `CGMapU256ToU256`); other fixed-width scalar maps show the same trend.

Type	Fun.	min	ave	max	const
`u256=>u256`	`get`	2982	2982	2982	
`CGMap[8]`	`get` (times)	76x	76x	76x	226288
`CGMap[64]`	`get` (times)	584x	584x	584x	1740456
`CGMap[256]`	`get` (times)	2327x	2327x	2327x	6939421
`u256=>u256`	`set`	22535	30660	45191	
`CGMap[8]`	`set` (times)	47x	35x	24x	1066475
`CGMap[64]`	`set` (times)	365x	268x	182x	8220485
`CGMap[256]`	`set` (times)	1455x	1070x	726x	32793532
`u64=>u64`	`get`	3012	3012	3012	
`CGMap[8]`	`get` (times)	73x	73x	73x	221190
`CGMap[64]`	`get` (times)	564x	564x	564x	1699462
`CGMap[256]`	`get` (times)	2249x	2249x	2249x	6775355
`u64=>u64`	`set`	22569	30438	44661	
`CGMap[8]`	`set` (times)	47x	35x	24x	1058061
`CGMap[64]`	`set` (times)	362x	268x	183x	8165119
`CGMap[256]`	`set` (times)	1443x	1070x	729x	32577270
`u8=>u8`	`get`	3012	3012	3012	
`CGMap[8]`	`get` (times)	75x	75x	75x	224886
`CGMap[64]`	`get` (times)	574x	574x	574x	1729030
`CGMap[256]`	`get` (times)	2289x	2289x	2289x	6893627
`u8=>u8`	`set`	22569	29783	44493	
`CGMap[8]`	`set` (times)	47x	36x	24x	1062855
`CGMap[64]`	`set` (times)	364x	276x	184x	8206897
`CGMap[256]`	`set` (times)	1451x	1099x	736x	32745768

Full-scan maps are expensive and scale linearly with public capacity. A capacity-256 map is not an economically viable replacement for a large on-chain ERC20 balance table. It is a precise baseline for small confidential maps and a motivation for ORAM-backed state.

Table 8: Admitted-artifact runtime snapshot from the proof-carrying toolchain.

Workload	Backend	Cap.	Gas	Reads	Writes	Read B	Write B	Trace KiB	Artifact KiB
`Fixed/classify`	full scan	–	4,096	1	1	124	124	9.3	8.6
`AuthScan/auth`	full scan	4	8,192	17	5	2,108	620	19.6	13.2
`Registry/clearIfMatch`	full scan	8	12,288	16	8	3,040	1,520	21.0	13.2
`Mixer/deposit`	full scan	8	16,384	18	9	3,288	1,644	23.5	15.7
`Mixer/deposit`	full scan	64	32,768	130	65	24,568	12,284	105.3	15.8
`Mixer/deposit`	Path ORAM	64	12,288	50	49	56,644	56,520	152.9	16.0
`Mixer/withdraw`	full scan	8	16,384	18	9	3,288	1,644	23.6	15.7
`Mixer/withdraw`	full scan	64	32,768	130	65	24,568	12,284	105.4	15.7
`Mixer/withdraw`	Path ORAM	64	12,288	50	49	56,644	56,520	153.0	16.0
`ERC20/transferFrom`	full scan	8+16	24,576	64	24	11,392	4,368	53.3	29.2
`ERC20/transferFrom`	Path ORAM	64+128	16,384	178	178	198,768	198,768	487.7	29.7

8.2 End-to-end toolchain results

The end-to-end toolchain evaluation uses five complete workloads: a fixed-slot classifier, an authentication scan, a registry, a confidential mixer, and an ERC20-like transfer contract. The raw configuration is a negative admission control: confidential state with a raw backend is rejected in proof-grade builds. Executable raw and gas-only variants are kept only as positive controls. Exact UO artifacts use cells, arrays, and full-scan maps. Path ORAM artifacts use the computational ORAM proof route; the rows in Table 8 use the bounded local-position-map route.

Security. Every admitted exact-CG artifact in the benchmark suite has equal opcode traces, logical accesses, logical writes, physical reads, and physical write traces for paired secret executions. Across the admitted exact-CG and ORAM leakage reports, all AUC values are 0.5 and all mutual-information estimates are 0 bits. By contrast, the raw-insecure and gas-only positive controls leak update-class information with maximum AUC 1.0 on every workload. The same suite reports functional equivalence for all five workloads.

Backend and scalability. The backend comparison is the main scalability lesson. As shown in Table 9, for a capacity-64 mixer, Path ORAM reduces modeled gas and physical operation count, but moves more bytes. This is a plausible tradeoff for a TEE-backed confidential chain where host I/O and proof objects matter differently from Ethereum gas.

Table 9: Same-capacity backend comparison for the mixer at capacity 64. Ratios are Path ORAM divided by full scan.

Workload	Gas ratio	Operation ratio	Byte ratio
`Mixer/deposit`	0.38×	0.51×	3.1×
`Mixer/withdraw`	0.38×	0.51×	3.1×

However, the `ERC20Lite` ORAM row (Table 8) should not be overread: while Coga can handle large ERC20 state, it is not evidence that current ORAM settings solve large ERC20 state cheaply. The private-position-map route is a proof-grade route for larger maps, its current one-access-per-map limitation means the multi-access `ERC20Lite` calls will be rejected, which may limit its adoption.

The proof-artifact size gate is the second scalability result. Table 10 compares the size of fully materialized ORAM witness families with the compact theorem inputs that the checker actually consumes. The single-access row is still small enough to check directly, but the mixer and ERC20-like rows would become unreviewable if every leaf assignment and fresh-coin witness were enumerated. The compact theorem-input path keeps the checked artifacts within the configured budget while preserving the same public theorem boundary.

Table 10: Path ORAM proof-artifact size gate. Large enumerated witness families are checked through compact theorem inputs rather than materialized.

Scenario	Estimated bytes	Checked bytes
`SingleAccessMap/put` pair	8.7×10^6	8.7×10^6
`Mixer/deposit` pair	3.3×10^{10}	1.49×10^5
`ERC20/transferFrom` pair	7.1×10^{18}	4.88×10^5

8.3 Engineering cost

The current benchmark programs are small, but they show where developer work goes. The Solidity source is unchanged between raw and admitted variants. The effort is in metadata: choosing a backend, declaring public capacities, assigning gas budgets, and marking declassifications.

Table 11: Engineering cost in the current benchmark set.

Program	Src	Sidecar	Main change
`FixedSlotClassifier`	15	27	raw cell → CG cell
`AuthScan`	21	32	CG array and result cell
`RegistryCRUD`	22	36	capacity-8 full-scan map
`ConfidentialMixer`	23	41	full-scan or ORAM map
`ERC20Lite`	40	55	balance and allowance maps

This is not a claim that arbitrary production contracts port with only a few metadata lines. Rather, the result shows that once a contract is written in the admitted style, the additional policy burden is explicit and auditable.

Table 12: Deployment guidance supported by the current design and evaluation.

Use case	Recommended backend	Guidance
Fixed scalar confidential state	UO cell	Good fit; exact trace equality and small physical schedule
Small arrays and bounded maps	full scan	Good fit when public capacity is small and byte traffic fits the application
Mixer-like maps around capacity 64	full scan or local Path ORAM	ORAM may reduce operations but increases byte traffic and proof assumptions
Large single-access confidential maps	private-position-map Path ORAM	Proof-grade route with compact symbolic evidence; one logical access per map per entrypoint
Large ERC20-like global maps	not solved cheaply	Multi-access maps exceed the private-position-map route with higher local Path ORAM cap
Encrypted client returns	`encrypted_to_client` return policy	Hides plaintext return from the host while exposing ciphertext evidence and session binding
Raw storage plus gas padding	not a defense	May equalize modeled or receipt gas while leaving M^+ host traces
Stock EVM clients	outside claim	Need a modeled confidential runtime and storage object layer

9 Deployment Guidance and Limitations

The value of constant-gas programming depends on choosing the right state shape. Table 12 summarizes the current guidance.

Three limitations deserve emphasis.

First, the threat model is strong but explicit. We defend against a host-visible marker-aware write observer. We do not defend against every TEE side channel. A deployment must still validate instruction-level timing, cache behavior, I/O behavior, attestation, rollback, freshness, crash recovery, and compiler/runtime version binding. The production acceptance theorem binds these items to package, proof, runtime, quote, return-channel, and counter evidence, but the underlying TEE guarantees remain assumptions.

Second, the language subset is restrictive. The current toolchain rejects dynamic data structures, external calls, events, custom errors, arbitrary reverts, inheritance, constructors, fallback functions, multiple returns, unsupported integer widths, and inline assembly. The current semantics profile deliberately uses wrapping `uint256` arithmetic, division or modulo by zero returning zero, canonical fixed-size ABI encoding, `require`/abort rollback, and only the admitted public context variables. Solidity 0.8 checked-arithmetic behavior, Solidity's division/modulo-by-zero revert behavior, full ABI semantics, arbitrary revert semantics, additional context variables, and value-level equivalence between the Lean hash abstraction and the runtime Keccak primitive remain outside the current theorem. This is a correctness choice. Adding those features requires modeling their public observations and proving that the runtime implements the model.

Third, large maps remain the hard problem. The full-scan `CGMap` design is exact but expensive. The ORAM design is promising but shifts cost into byte traffic, metadata, cryptographic assumptions, position-map management, fresh-coin evidence, and proof complexity. The local-position-map route supports multi-access entrypoints only up to capacity 128, while the scalable private-position-map route currently supports only one logical access per map per entrypoint. For many applications, the right answer today is to redesign the confidential state so that the secret-indexed part is small, batched, or off the critical path.

10 Related Work

Constant-time programming and verification. Constant-time programming originates from timing attacks on cryptographic implementations [8] and has been formalized through control-flow and memory-observation models [1, 2, 11]. Compiler work such as Constantine and related control-flow linearization systems show that automated transformations can remove many control and address channels [3, 17, 18]. Constant-gas programming keeps that foundation but adds update classes to the observation.

TEE side channels and TEE-oriented program transformation. Prior work shows that SGX and confidential-VM systems expose channels beyond plain timing [10, 22, 23]. DOVE, TrustJS, and SGX-BigMatrix demonstrate the broader systems pattern of adapting programs or runtimes to TEE deployment constraints [7, 9, 16]. Our contribution is specific to confidential smart contracts: the EVM storage/gas model makes update classes security relevant even for code that is already constant-time.

ORAM and oblivious storage. ORAM hides logical access locations [20]. This is necessary for secret-indexed mappings, but not sufficient under M^+. The physical ORAM buckets, filler blocks, position map, and metadata writes must themselves avoid marker-aware update leakage. Coga therefore treats ORAM as one backend inside a larger update-oblivious storage discipline.

Smart-contract gas and privacy. Ethereum gas metering has been refined through EIPs that price storage updates, cold/warm accesses, and refunds [4, 5, 21]. These mechanisms are economically motivated, but in confidential smart contracts they become part of the side-channel surface. The contribution of this paper is to treat storage-update classes as explicit side-channel observations in confidential EVMs, formalize the resulting M^+ security condition, validate the concern on a Sapphire-style confidential stack, and connect the programming discipline to a proof-carrying toolchain boundary.

11 Conclusion

Constant-time programming hides secret-dependent control flow and memory addresses. Confidential smart contracts need one more guarantee: secret data must not influence the semantic class of persistent storage updates. This paper presents constant-gas programming as that guarantee through a coherent explanation on why constant-time Solidity is insufficient, how a confidential EVM can leak even after gas padding, what a precise M^+ security definition should say, and how a restricted but checked toolchain, Coga, can rigorously and practically enforce the property.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 53–70.

[2] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, 462–476. doi:10.1145/3460120.3484761

[3] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, 715–733. doi:10.1145/3460120.3484583

[4] Vitalik Buterin and Martin Swende. 2020. EIP-2929: Gas cost increases for state access opcodes. Ethereum Improvement Proposals, no. 2929. https://eips.ethereum.org/EIPS/eip-2929 Accessed 2026-06-25.

[5] Vitalik Buterin and Martin Swende. 2021. EIP-3529: Reduction in refunds. Ethereum Improvement Proposals, no. 3529. https://eips.ethereum.org/EIPS/eip-3529 Accessed 2026-06-25.

[6] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained*. Technical Report 2016/086. IACR Cryptology ePrint Archive. 1–118 pages. https://eprint.iacr.org/2016/086

[7] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. 2017. TrustJS: Trusted Client-side Execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec '17)*. Association for Computing Machinery, Article 7, 7:1–7:6 pages. doi:10.1145/3065913.3065917

[8] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO '96 (Lecture Notes in Computer Science, Vol. 1109)*. Springer, 104–113. doi:10.1007/3-540-68697-5_9

[9] Hyun Bin Lee, Tushar M. Jois, Christopher W. Fletcher, and Carl A. Gunter. 2021. DOVE: A Data-Oblivious Virtual Environment. In *28th Annual Network and Distributed System Security Symposium (NDSS 2021)*. The Internet Society. doi:10.14722/ndss.2021.23056

[10] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 717–732.

[11] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology – ICISC 2005 (Lecture Notes in Computer Science, Vol. 3935)*. Springer, 156–168. doi:10.1007/11734727_14

[12] Oasis Protocol Foundation. [n. d.]. Sapphire.sol: Sapphire Contracts Library, padGas. Oasis API documentation. https://api.docs.oasis.io/sol/sapphire-contracts/contracts/Sapphire.sol/library.Sapphire.html Accessed 2026-06-25.

[13] Oasis Protocol Foundation. 2026. Sapphire: Concepts. Oasis documentation. https://docs.oasis.io/build/sapphire/develop/concept/ Accessed 2026-06-25.

[14] Oasis Protocol Foundation. 2026. Sapphire ParaTime. Oasis documentation. https://docs.oasis.io/build/sapphire/ Accessed 2026-06-25.

[15] Oasis Protocol Foundation. 2026. Sapphire: Security. Oasis documentation. https://docs.oasis.io/build/sapphire/develop/security/ Accessed 2026-06-25.

[16] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework with Trusted Processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, 1211–1228. doi:10.1145/3133956.3134095

[17] Luigi D. C. Soares, Michael Canesche, and Fernando Magno Quintão Pereira. 2023. Side-channel Elimination via Partial Control-flow Linearization. *ACM Transactions on Programming Languages and Systems* 45, 2, Article 13 (June 2023), 13:1–13:43 pages. doi:10.1145/3594736

[18] Luigi D. C. Soares and Fernando Magno Quintão Pereira. 2021. Memory-Safe Elimination of Side Channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 200–210. doi:10.1109/CGO51591.2021.9370305

[19] Solidity Project. [n. d.]. Layout of State Variables in Storage and Transient Storage. Solidity documentation. https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html Accessed 2026-06-25.

[20] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4, Article 18 (April 2018), 26 pages. doi:10.1145/3177872

[21] Wei Tang. 2019. EIP-2200: Structured Definitions for Net Gas Metering. Ethereum Improvement Proposals, no. 2200. https://eips.ethereum.org/EIPS/eip-2200 Accessed 2026-06-25.

[22] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, 2421–2434. doi:10.1145/3133956.3134038

[23] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. 2023. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6789–6806.

A Oasis Workload Contract

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.21;

contract WrappedIdenticalBytecodeP1 {
    address internal constant PAD_GAS =
        0x010000000000000000000000000000000000000a;

    uint256 public slotCount;
    uint64 public padTo;

    constructor(uint256 slotCount_, uint64 padTo_) {
        slotCount = slotCount_;
        padTo = padTo_;
    }

    function measurePadded(bytes32 token) external {
        uint256 value = _classify(token);
        _writeAll(value);
        _padGas(padTo);
    }

    function measureUnpadded(bytes32 token) external {
        uint256 value = _classify(token);
        _writeAll(value);
    }

    function computeOnlyPadded(bytes32 token) external
      returns (bytes32 acc)
    {
        uint256 value = _classify(token);
        uint256 n = slotCount;
        assembly ("memory-safe") {
            let ptr := mload(0x40)
            acc := 0
            for { let i := 0 } lt(i, n) { i := add(i, 1) } {
                mstore(ptr, acc)
                mstore(add(ptr, 0x20), i)
                mstore(add(ptr, 0x40), value)
                acc := keccak256(ptr, 0x60)
            }
        }
        _padGas(padTo);
    }

    function _classify(bytes32 token) internal pure
      returns (uint256 value)
    {
        assembly ("memory-safe") {
            let isOne := iszero(xor(token,
            0x2222222222222222222222...2222222222222222222222))
            let isOneAlt := iszero(xor(token,
            0x3333333333333333333333...3333333333333333333333))
            value := or(isOne, isOneAlt)
        }
    }

    function _writeAll(uint256 value) internal {
        uint256 n = slotCount;
        assembly ("memory-safe") {
            for { let i := 0 } lt(i, n) { i := add(i, 1) } {
                sstore(add(0x1000, i), value)
            }
        }
    }

    function _padGas(uint128 amount) internal {
```

```
        (bool success, ) = PAD_GAS.call(abi.encode(amount));
        require(success, "padgas failed");
    }
}
```

B Complete set function for constant-gas map

While the completed `CGMap.set` function, presented below, is less important in illustrating scientific principles presented in this paper, it is nevertheless still relevant to understand the full implementation details of the constant-gas replacement.

```
function set(uint256 k, uint256 v) public returns (bool) {
  bool k_is0 = (k == 0);
  bool v_is0 = (v == 0);

  uint256 k_in = cg_select_u256(k_is0, MARK_ZERO, k);
  uint256 v_in = cg_select_u256(v_is0, MARK_ZERO, v);

  uint8 status_empty = u8_set_bit(
    u8_set_bit(STATUS_BASE, STATUS_BIT_K_IS_ZERO),
    STATUS_BIT_V_IS_ZERO
  );

  uint8 status_key_zero = u8_set_bit(
    STATUS_BASE,
    STATUS_BIT_K_IS_ZERO
  );

  uint256 next = version + 1;

  // Iteration 1: try to update or delete an existing entry.
  bool updated = false;
  for (uint16 i = 0; i < CAPACITY; i++) {
    Entry storage entry = entries[i];

    uint256 ek = entry.key[version];
    uint256 ev = entry.value[version];
    uint8 es = entry.status[version];

    bool ek_is0 = u8_is_bit_set(es, STATUS_BIT_K_IS_ZERO);
    bool ev_is0 = u8_is_bit_set(es, STATUS_BIT_V_IS_ZERO);
    uint256 ek_out = cg_select_u256(ek_is0, 0, ek);

    uint256 ek_next = ek;
    uint256 ev_next = ev;
    uint8 es_next = es;

    bool p_context = true;

    // if (updated) {
    bool _p0 = updated && p_context;
    //   // already updated, do nothing
    // } else {
    bool p1 = (!updated) && p_context;

    //   if (ev_is0) {
    bool _p2 = ev_is0 && p1;
    //     // empty entry, cannot update
    //   } else {
    bool p3 = (!ev_is0) && p1;

    //     if (k_is0) {
    bool p4 = k_is0 && p3;

    //       if (ek_is0) {
    bool p5 = ek_is0 && p4;

    //         if (v_is0) {
    bool p6 = v_is0 && p5;
    //           // delete the slot
    ek_next = cg_select_u256(p6, MARK_ZERO, ek_next);
    ev_next = cg_select_u256(p6, MARK_ZERO, ev_next);
    es_next = cg_select_u8(p6, status_empty, es_next);

    //         } else {
    bool p7 = (!v_is0) && p5;
    //           // update the value
    ev_next = cg_select_u256(p7, v_in, ev_next);

    //         }

    //         // in both cases, set the updated flag
    updated = p5 || updated;

    //       } else {
    bool _p8 = (!ek_is0) && p4;
    //         // key == 0 != entry_key, do nothing
    //       }

    //     } else {
    bool p9 = (!k_is0) && p3;

    //       if (ek_is0) {
    bool _p10 = ek_is0 && p9;
    //         // key != 0 == entry_key, do nothing
    //       } else {
    bool p11 = (!ek_is0) && p9;

    //         if (k == ek_out) {
    bool p12 = (k == ek_out) && p11;

    //           if (v_is0) {
    bool p13 = v_is0 && p12;
    //             // delete the slot
    ek_next = cg_select_u256(p13, MARK_ZERO, ek_next);
    ev_next = cg_select_u256(p13, MARK_ZERO, ev_next);
    es_next = cg_select_u8(p13, status_empty, es_next);

    //           } else {
    bool p14 = (!v_is0) && p12;
    //             // update the value
    ev_next = cg_select_u256(p14, v_in, ev_next);
    //           }

    //           // in both cases, set the updated flag
    updated = p12 || updated;

    //         } else {
    bool _p15 = (k != ek_out) && p11;
    //           // key != entry_key, do nothing
    //         }
    //       }
    //     }
    //   }
    // }

    entry.key[next] = ek_next;
    entry.value[next] = ev_next;
    entry.status[next] = es_next;
  }

  // Catch up the public version index after the update/delete pass.
  version = next;
  next = version + 1;

  // Iteration 2: try to insert into the first empty entry.
  bool inserted = false;
  for (uint16 i = 0; i < CAPACITY; i++) {
    Entry storage entry = entries[i];

    uint256 ek = entry.key[version];
    uint256 ev = entry.value[version];
    uint8 es = entry.status[version];

    bool ev_is0 = u8_is_bit_set(es, STATUS_BIT_V_IS_ZERO);

    uint256 ek_next = ek;
    uint256 ev_next = ev;
    uint8 es_next = es;

    bool p0 = true;

    // if (updated) {
    bool _p1 = updated && p0;
    //   // already updated, do nothing
    // } else {
    bool p2 = (!updated) && p0;

    //   if (inserted) {
```

```
    bool _p3 = inserted && p2;
    //     // already inserted, do nothing
    //   } else {
    bool p4 = (!inserted) && p2;

    //     if (ev_is0) {
    bool p5 = ev_is0 && p4;

    //       if (v_is0) {
    bool _p6 = v_is0 && p5;
    //         // setting an absent key to zero: no insertion
    //       } else {
    bool p7 = (!v_is0) && p5;
    //         // insert here
    ek_next = cg_select_u256(p7, k_in, ek_next);
    ev_next = cg_select_u256(p7, v_in, ev_next);
    es_next = cg_select_u8(
      p7,
      cg_select_u8(k_is0, status_key_zero, STATUS_BASE),
      es_next
    );
    inserted = p7 || inserted;
    //       }

    //     } else {
    bool _p8 = (!ev_is0) && p4;
    //       // occupied entry, do nothing
    //     }
    //   }
    // }

    entry.key[next] = ek_next;
    entry.value[next] = ev_next;
    entry.status[next] = es_next;
  }

  // Commit the insertion/copy-through pass.
  version = next;

  // True means the logical assignment succeeded.
  // If v == 0 and the key was absent, the assignment is a no-op and succeeds.
  // If v != 0 and the key was absent, failure means the bounded map was full.
  return updated || inserted || v_is0;
}
```