STARKWARE

Smart Contract Audit Report

Conducted by CryptoExperts

As part of our due process, we retained CryptoExperts to review the design document and related source code of the Cairo & SHARP verifiers. We chose to work with CryptoExperts based on our good experience and interaction with them in the past.

We are happy to share the key findings below, followed by the full report.

Vulnerability Severity Classification

The current version of the report is an update of our original report after counterauditing modifications from StarkWare.

Each observation is appended with a status:
Resolved (✓), Partially resolved (~), Unresolved (✗).

Most of our recommendations have been addressed. Unresolved or partially resolved issues are related to documentation or coding practices which are of minor importance.

Category	Number of findings	✓	~	✗
High risk	-	-	-	-
Medium risk	-	-	-	-
Low risk	1	1	-	-
Coding Practices	8	6	-	2
Documentation	3	2	1	-
Total	12	9	1	2

Contents

1 Introduction

StarkWare is a company developing scalability and privacy technologies for blockchain applications. In particular, StarkWare develops a STARK-powered level-2 scalability engine which uses cryptographic proofs to attest to the validity of a batch of transactions. As part of StarkWare's technology, the Cairo STARK-friendly CPU architecture allows developers to easily write STARK-provable programs for general computation.

In this context, CryptoExperts has conducted a first audit of StarkWare's STARK verifier (implemented as a Solidity smart contract) in late 2021. StarkWare was then willing to perform a follow-up audit of their Cairo verifier and associated shared proof service (SHARP), which are built on top of the STARK verifier.

Upon business agreement with StarkWare, CryptoExperts has conducted this second audit in March and April 2022. The primary goal of this audit was to ensure that the Solidity code of the Cairo and SHARP verifiers (together with the Cairo code of the bootloader) correctly verify the computational integrity of Cairo programs. The service consisted in a study of the Cairo specification and a review of the code by two engineers, experts in cryptography.

The present report contains the results of this audit. The curent version of the report (v2.0) is an update of the original report with the subsequent audit of the general bootloader (extending the simple bootloader) which has been conducted in July 2022.

We first give an overview of the code (Section 1.1) and a summary of the audit methodology and findings (Section 1.2). We provide some complementary documentation of the Cairo verifier implementation in Section 2 then present our review of this implementation in Section 3. This review includes a summary of the functionality of each smart contract together with a list of observations and recommendations. We then provide some complementary documentation of the SHARP verifier implementation in Section 4, and its review in Section 5.

1.1 Overview of the code

The scope of the audit includes the Cairo Verifier and SHARP verifier, a.k.a. GPS (Generic Proof Service) verifier, which are both built on top of the STARK verifier previously audited [1].

The first audited code implements the Cairo verifier. It is composed of the 8 source files at the root of the following GitHub folder:

`https://github.com/starkware-libs/starkex-contracts/tree/master/evm-verifier/solidity/contracts/cpu`

The audited version of the code corresponds to the commit `0efa9ce`, "StarkEx v4.0", from October 14th, 2021. Those 8 source files contain about 800 lines of code written in Solidity language (excluding `CairoBootloaderProgram.sol` which is mainly auto-generated).

The second audited code implements the SHARP verifier, a.k.a. GPS (Generic Proof Service) verifier. The latter verifies the execution of a specific Cairo program, the *boot-*

loader, which is also in the scope of the audit. The audited Solidity code is composed of the 2 source files at the following GitHub repository:

`https://github.com/starkware-libs/starkex-contracts/tree/master/evm-verifier/solidity/contracts/gps`

The audited version of the code corresponds to the commit `0efa9ce`, "StarkEx v4.0", from October 14th, 2021. Those 2 source files contain about 600 lines of code written in Solidity language.

The audited Cairo code is composed of the simple bootloader and the (general) bootloader. The simple bootloader is composed of the 3 source files at the following GitHub repository:

`https://github.com/starkware-libs/cairo-lang/tree/master/src/starkware/cairo/bootloaders/simple_bootloader`

The audited version of the code corresponds to the commit `4e23351`, "Cairo v0.8.0", from March 13th, 2022. Those 3 source files contain about 500 lines of code written in Cairo language. The (general) bootloader is composed of the following source file:

`https://github.com/starkware-libs/cairo-lang/blob/master/src/starkware/cairo/bootloaders/bootloader/bootloader.cairo`

The audited version of the code corresponds to the commit `167b28b`, " Cairo v0.9.0.", from June 5th, 2022. This source file contains about 350 lines of code written in Cairo language.

The different smart contracts are represented in Figures 1 and 2 with their inheritance relations. The contracts in pink have been audited [1]. The contracts in green are out of the scope of the present audit. The contracts in yellow are auto-generated and out of the scope of the audit.

1.2 Methodology and summary of findings

The main goal of this audit was to validate the soundness of the reviewed implementation of the Cairo and SHARP verifiers. More precisely, this audit aims

- to confirm that the implemented verification process is compliant to the specification of the Cairo and SHARP verifiers,
- to check the absence of flaw in the implementation which would allow an adversary to forge a valid proof for an invalid statement (with less effort than the target security level).

The audit methodology consisted in an in-depth review of the code by two different persons (engineers, junior and senior experts in cryptography), confronting our understanding of the code and keeping track of our observations.

Our observations are categorized as follows:

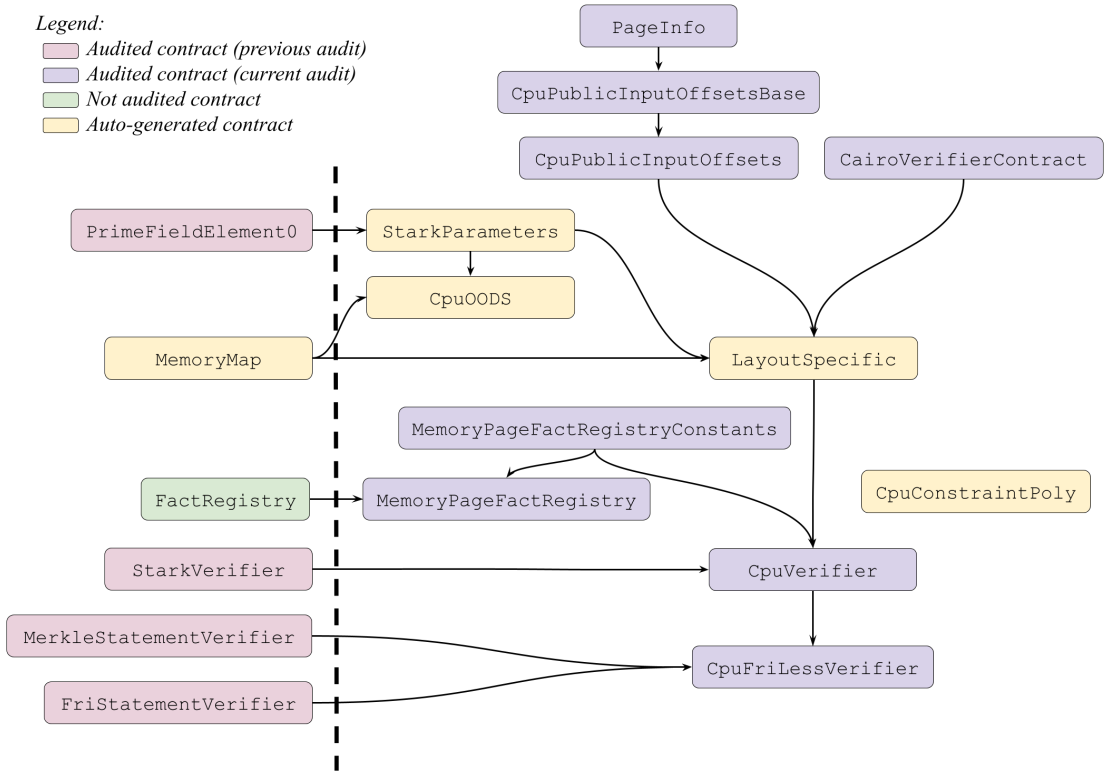

Figure 1: The inheritance relations between the smart contracts of the Cairo verifier.

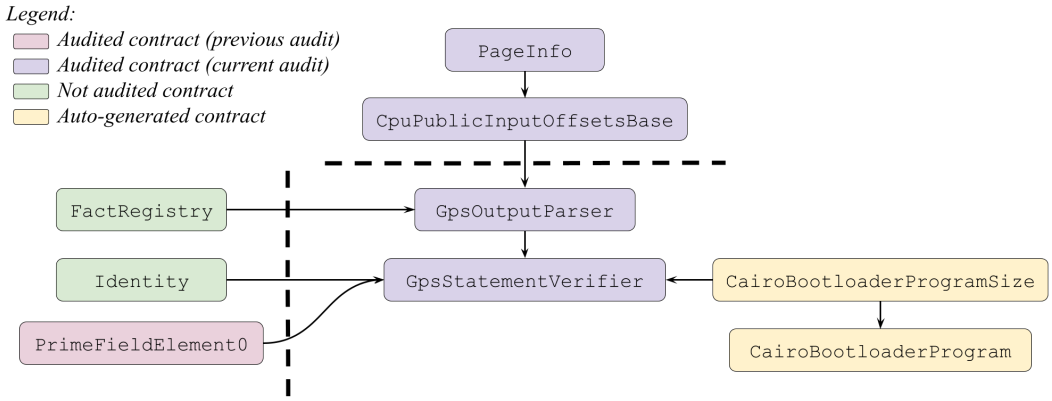

Figure 2: The inheritance relations between the smart contracts of the SHARP verifier.

- Observations that may impact the soundness of the verifier, rated as
  - high risk (flagged ●),

  - medium risk (flagged ●),
  - low risk (flagged ●).

- Observations related to coding practices and implementation choices (flagged ●).
  - These observations do not translate into a direct risk on the soundness of the verifier but addressing them would make the code clearer, more efficient and/or less prone to errors.
- Observations related to documentation, comments, variable naming (flagged ■).
  - These observations do not translate into a direct risk on the soundness of the verifier but addressing them would facilitate the understanding of the code by third parties (users, developers, auditors).

Each observation comes with an associated recommendation to fix or improve the underlying issue.

General remark. We were provided with a clear set of documentations about Cairo [2, 5] but sometimes missed detailed specifications for the reviewed implementation. To reach a global and confident understanding of the implementation, we wrote down the missing specifications according to our understanding of the code, which we include in the report (see Section 2 for the Cairo verifier and Section 4 for the SHARP verifier).

Summary of findings on the Cairo verifier. Our findings are summarized in the table below. Our observations are mainly related to coding practices and documentation. One observation relates to a test which might be missing but the associated risk is considered low. We therefore conclude that, up to the limitations inherent to human code review, the code is a sound implementation of the Cairo verifier.

The current version of the report is an update of our original report after counter-auditing modifications from StarkWare. Each observation is appended with a status: Resolved (✓), Partially resolved (∼), Unresolved (✗). Most of our recommendations have been addressed. Unresolved or partially resolved issues are related to documentation or coding practices which are of minor importance. The updated source code corresponds to the commit `f81ba5f` ("SHARP EVM Verifier v3.0", from December 8th, 2022) of the GitHub folder:

`https://github.com/starkware-libs/starkex2.0-contracts/tree/master/evm-verifier/solidity/contracts/cpu`.

Category	Number of findings	✓	∼	✗
● High risk	0	0	0	0
● Medium risk	0	0	0	0
● Low risk	1	1	0	0
● Coding practices	8	6	0	2
■ Documentation	3	2	1	0
Total	**12**	**9**	**1**	**2**

Summary of findings on the SHARP verifier and bootloader. Our findings are summarized in the table below. Our observations are only related to coding practices and documentation. We therefore conclude that, up to the limitations inherent to human code review, the code is a sound implementation of the SHARP verifier and associated bootloader.

The current version of the report is an update of our original report after counter-auditing modifications from StarkWare. Each observation is appended with a status: Resolved (✓), Partially resolved (∼), Unresolved (✗). Most of our recommendations have been addressed. Unresolved issues are related to documentation or coding practices which are of minor importance. The updated source code for the SHARP verifier corresponds to the commit `f81ba5f` ("SHARP EVM Verifier v3.0", from December 8th, 2022) of the GitHub folder:

`https://github.com/starkware-libs/starkex2.0-contracts/tree/master/evm-verifier/solidity/contracts/gps`.

The updated source code for the bootloader corresponds to the commit `de741b9` ("Cairo v0.10.3.", from December 2nd, 2022) of the GitHub folder:

`https://github.com/starkware-libs/cairo-lang/tree/master/src/starkware/cairo/bootloaders`.

Category	Number of findings	✓	∼	✗
High risk	0	0	0	0
Medium risk	0	0	0	0
Low risk	0	0	0	0
Coding practices	3	2	1	0
Documentation	5	5	0	0
Total	**8**	**7**	**1**	**0**

2 Documentation for the Cairo verifier

In this report, we use the notations and terminology introduced in

- the article [2] which presents Cairo,
- the ethSTARK documentation [3],
- the code review [1] of the STARK verifier.

The present section provides some documentation for the audited Cairo verifier implementation which completes the above references.

2.1 The CPU architecture

The Cairo article [2] describes the Cairo machine which behaves like a central processing unit (CPU) executing basic instructions. The instruction set has been designed such that there exists an efficient *algebraic intermediate representation* (AIR) of the Cairo machine processing: a loop fetching an instruction, decoding and executing that instruction, and continuing to the next instruction. The Cairo machine has three registers:

- the *program counter*, denoted `pc`, which contains the memory address of the next Cairo instruction to be executed,
- the *allocation pointer*, denoted `ap`, which (by convention) points to the first memory cell that has not been used by the program so far, and
- the *frame pointer*, denoted `fp`, which (by convention) points to the beginning of the stack frame of the current function (see Section 2.4).

Moreover, the CPU has access to a *nondeterministic continuous read-only memory* $\mathfrak{m}$. To represent a value of the memory at the address a, we use $\mathfrak{m}(a)$ or $[a]$. Since the memory is read-only, we can write without ambiguity that an address a has a value v, *i.e.* $\mathfrak{m}(a) = v$, since the value cannot change.

The address space and value range are both defined as the field $\mathbb{F}_p$ used for the underlying AIR (see *e.g.* [3, 1]), hence the memory can be seen as a function $\mathfrak{m} : \mathbb{F}_p \to \mathbb{F}_p$.

2.2 Builtins

The Cairo machine further relies on *builtins* which offer a way to efficiently perform some specific computation tasks. For example, a call to the Pedersen builtin enables the Cairo developer to compute the Pedersen hash digest of two field elements without needing to implement this functionality in Cairo syntax. The idea is similar to the addition of optimized low-level execution units (a.k.a. co-processors) to a physical CPU.

A builtin aims to perform a specific computation which involves several field elements. A builtin *instance* is a set of those field elements and the *size* of the instance is defined as

the size of this set. For example, the Pedersen builtin aims to check that $z = \text{Hash}(x, y)$ where x, y and z are three field elements. In this case, the size of the builtin instance is three.

Each builtin is assigned to a memory segment which is split into small chunks. Each chunk corresponds to a builtin instance, and the values stored on a chunk must satisfy the relation of the builtin. In the case of the Pedersen builtin, the memory segment has the form

	a	$a+1$	$a+2$	$a+3$	$\ldots$					
$\mathfrak{m}(\cdot)$	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	$\ldots$

where a is the first address of the memory segment, and the AIR constraints will impose that the relation

$$z_i = \text{Hash}(x_i, y_i)$$

is verified for all i. In practice, to use this builtin in a Cairo program, the developer simply needs to assert $\mathfrak{m}(a+3i) = x_i$ and $\mathfrak{m}(a+3i+1) = y_i$ for some i, and then she can consider that $\mathfrak{m}(a+3i+2)$ contains the hash digest $z_i = \text{Hash}(x_i, y_i)$.

In order to use a builtin, a Cairo developer needs to know the address of the first unused instance. To proceed, she will always store and update the pointer to this address. We call this pointer the *builtin pointer*. The number of builtin instances is limited by the size of the memory segment allocated to the builtin. This number is given by the number T of state transitions (see next section) and a parameter r called *builtin ratio* via the formula

$$r = \frac{T}{\texttt{max\_nb\_instances}} .$$

The audited version of Cairo includes the five following builtins:

- Output (to output data from the Cairo program),
- Pedersen (to compute a Pedersen hash of two field values),
- Range-check (to check a value belong to some predefined range),
- ECDSA (to verify an ECDSA signature),
- Bitwise (to compute the bitwise AND, XOR, OR between two field values).

2.3 Execution of a program

A state of the Cairo machine is a triple of register values $(\texttt{pc}, \texttt{ap}, \texttt{fp})$. The execution of a program can thus be represented as a sequence of states $(\texttt{pc}_i, \texttt{ap}_i, \texttt{fp}_i)_i$ and a memory function $\mathfrak{m} : \mathbb{F}_p \to \mathbb{F}_p$ (which to each address a accessed by the program associates a value v such that $\mathfrak{m}(a) = v$). The transition between two states is defined by the instruction pointed by the $\texttt{pc}$ register:

$$(\texttt{pc}_i, \texttt{ap}_i, \texttt{fp}_i) \xrightarrow{\mathfrak{m}(\texttt{pc}_i)} (\texttt{pc}_{i+1}, \texttt{ap}_{i+1}, \texttt{fp}_{i+1}) .$$

We denote T the number of steps (or state transitions) in the program execution.

The execution trace which is the input statement of the STARK verifier is a large two-dimensional array of W columns and L rows. The number of columns W is defined by the Cairo layout and depends on the used builtins (see hereafter). Each state of the Cairo machine is represented by a small set of successive rows called *component*. We denote `CPU_COMPONENT_HEIGHT` the number of rows in a component. The trace length L (*i.e.* its total number of rows) is then given by

$$L := \texttt{CPU\_COMPONENT\_HEIGHT} \cdot T.$$

In a component, we can find the current values of the registers `pc`, `ap` and `fp`. We can also find all the intermediate variables of the state transition described in Section 4.5 of [2]. Let us remark that the final state (*i.e.* the state of the Cairo machine after T steps) is not represented in the execution trace. The AIR constraints check that the last instruction is well executed (and that the resulting memory mapping is correct) but they do not check the final values of the three registers (`pc`, `ap`, `fp`).

We call *layout* the organization of all the variables in the execution trace. A layout depends on which builtins are supported, but it also depends on the configuration of those builtins (how many times the builtins can be used, *i.e.* the building ratio, the size of its instances, etc.). Defining a layout consists in choosing a trade-off between universality (*i.e.* the scope of the Cairo programs which can use this layout) and performances (*e.g.* minimal number of trace cells). Indeed, improving the universality of a layout implies increasing the number of constraints and the number of columns in the execution trace.

The audited source code includes four different layouts. Table 1 summarizes the parameters of those layouts. To illustrate the aforementioned trade-off, let us compare Layout 1 and Layout 2. Both support the same builtins though Layout 2 is cheaper to verify: it has only 10 columns, compared to 22 columns for Layout 1. On the other hand, the Cairo programs supported by Layout 1 can make more calls to the builtins compared to the programs supported by Layout 1 (see the corresponding builtin ratios).

As an illustration, Table 2 describes the structure of a component for Layout 1. All the notations are introduced in Section 9.10 "List of constraints" of the Cairo article [2]. The columns which are not described in Table 2 correspond to intermediate variables dedicated to the builtins.

The 17th column contains all the memory accesses during the program execution, where an odd row contains the memory address while the next even row contains the associated value. The 18th column contains the same list of memory accesses but sorted with increasing memory addresses. These two columns correspond to the lists L_1 and L_2 described in Sections 9.7 and 9.8 of the Cairo article [2]. The AIR constraints impose that L_2 is a permutation of L_1 (same pairs address-value with same occurrences) which is a continuous (the addresses are consecutive) and read-only (all the memory accesses to a given address read the same value). In practice, L_2 is further *padded* as explained in Section 2.5.2.

The 21st column contains the cumulative products (p^{m} for memory and p^{rc} for range-check) which are added to the trace after the interaction step (see Section 9.8 of the Cairo paper [2]).

	Layout 0	Layout 1	Layout 2	Layout 3
`Nb of interaction elements`	3	3	3	6
`Mask Size` M_1	201	200	128	291
`Number of rows involved in mask`	81	82	79	164
`Number` W `of columns`	25	22	10	27
`Nb of columns before interaction`	23	21	9	24
`Nb of columns after interaction`	2	1	1	3
`Public Memory Step`	8	8	8	16
`Pedersen Builtin Ratio`	8	8	32	8
`Range-check Builtin Ratio`	8	8	16	8
`ECDSA Builtin Ratio`	512	512	2048	512
`Bitwise Builtin Ratio`	-	-	-	256

Table 1: Parameters of the implemented layouts.

2.4 The main function

The entry point of a Cairo program is always the function `main`, from which Cairo supports (recursive) function calls. The Cairo paper suggests to implement a function call stack in the following way:

> *The frame pointer register (`fp`) points to the current frame in the "call stack". As you will see, it is convenient not to define `fp` as the beginning of the frame but rather as the beginning of the local variables' section, in a similar way to the behavior of the stack in common architectures. Each frame consists of four parts (`fp` refers to the current frame):*
>
> 1. *The arguments of the function provided by the caller. For example, `[fp - 3]`, `[fp - 4]`, ...*
> 2. *Pointer to the caller function's frame. Located at `[fp - 2]`.*
> 3. *The address of the instruction to be executed once the function returns (the instruction following the call instruction). Located at `[fp - 1]`.*
> 4. *Local variables allocated by the function. For example, `fp`, `fp + 1`, ...*
>
> *In addition, the return values of the function are placed in the memory at `[ap - 1]`, `[ap - 2]`, ... , where `ap` is the value of the ap register at the end of the function.*
>
> – Cairo article [2]

To parse the program input, a Cairo developer shall use *hints*. Hints are pieces of code that are inserted between Cairo instructions where additional work is required by the prover (see Section 2.5 of [2]). The program inputs are available in the hint scope thanks to the

	0	...	17	18	19	...	21
0	$\tilde{f}_0$	...	`pc`	a'^{m}	$\tilde{\mathtt{off}}_{\mathtt{dst}}$	...	p^{m}
1	$\tilde{f}_1$	...	`inst`	v'^{m}	`ap`	...	p^{rc}
2	$\tilde{f}_2$	...	0 (*p.m.*)	a'^{m}	a'^{rc}	...	p^{m}
3	$\tilde{f}_3$	...	0 (*p.m.*)	v'^{m}	t_0	...	
4	$\tilde{f}_4$	...	`op0_addr`	a'^{m}	$\tilde{\mathtt{off}}_{\mathtt{op1}}$	...	p^{m}
5	$\tilde{f}_5$	...	`op0`	v'^{m}	`mul`	...	p^{rc}
6	$\tilde{f}_6$	...	*builtin addr*	a'^{m}	a'^{rc}	...	p^{m}
7	$\tilde{f}_7$	...	*builtin value*	v'^{m}		...	
8	$\tilde{f}_8$	...	`dst_addr`	a'^{m}	$\tilde{\mathtt{off}}_{\mathtt{op0}}$	...	p^{m}
9	$\tilde{f}_9$	...	`dst`	v'^{m}	`fp`	...	p^{rc}
10	$\tilde{f}_{10}$	...	0 (*p.m.*)	a'^{m}	a'^{rc}	...	p^{m}
11	$\tilde{f}_{11}$	...	0 (*p.m.*)	v'^{m}	t_1	...	
12	$\tilde{f}_{12}$	...	`op1_addr`	a'^{m}		...	p^{m}
13	$\tilde{f}_{13}$	...	`op1`	v'^{m}	`res`	...	p^{rc}
14	$\tilde{f}_{14}$	...		a'^{m}	a'^{rc}	...	p^{m}
15	$\tilde{f}_{15}$	...		v'^{m}		...	
			L_1	L_2			

Table 2: Extract of the structure of a component for Layout 1.

Python dictionary `program_input`. The function `main` does not take a user-defined input. The only arguments of this function are *builtin pointers*, like in the following example:

```
%builtins output pedersen

from starkware.cairo.common.cairo_builtins import HashBuiltin
from starkware.cairo.common.hash import hash2

# Implicit arguments: addresses of the output and pedersen builtins.
func main{output_ptr, pedersen_ptr : HashBuiltin*}():
    ...

    # output_ptr and pedersen_ptr will be implicitly returned.
    return ()
end
```

In the above example, although they are implicit, the function `main` takes two arguments: the output builtin pointer and the Pedersen builtin pointer. In a similar way, the returned values of `main` do not correspond to the program output but to the updated builtin pointers, *i.e.* the new position of the first unused instance of each builtin. The program output is returned through the output builtin.

At the start of the execution, the registers `pc` and `ap` are fixed to some values by definition of the Cairo machine. In practice, the register `pc` is always initialized with the address 1. Let us stress that the address 0 is kept for the `null` pointer (*i.e.* $\mathtt{m}(0)$ is not

defined). Then, the register `fp` is initialized with the same value as `ap`. While compiling a Cairo program, the compiler add the following instructions at the beginning of the memory (*i.e.* at address 1 pointed by `pc` at start):

```
ap += n_args; call main; jmp rel 0.
```

The second instruction calls the function `main`, where the label `main` is replaced by the absolute position of the function in the bytecode. The last instruction is an infinite loop which does not change the register state:

$$(\mathtt{pc}_F, \mathtt{ap}_F, \mathtt{fp}_F) \xrightarrow{\texttt{jmp rel 0}} (\mathtt{pc}_F, \mathtt{ap}_F, \mathtt{fp}_F).$$

Due to those instructions, the final program counter $\mathtt{pc}_F$ is `5` (the three above instructions are respectively stored on two memory cells at addresses 1, 3, and 5).

2.5 Public memory

The Cairo verifier aims to check that a prover knows an execution trace of a given program on the Cairo machine. Such an execution trace involves a memory mapping (let us recall that the memory of the Cairo machine is read-only). Some values of this memory mapping are publicly known, *i.e.* are part of the statement to be checked. For example, the memory chunk which contains the program bytecode is usually public. This public part of the memory is called the *public memory* which is denoted $\mathfrak{m}^*$ in the Cairo documentation. The public memory can be thought of as a restriction of the memory $\mathfrak{m}$ to a domain $A^* \subset \mathbb{F}_p$ (the addresses of the public memory cells). The verifier must check that the execution trace committed by the prover is consistent with $\mathfrak{m}^*$.

2.5.1 Pages of the public memory

The public memory $\mathfrak{m}^*$ can be represented as a list $\{(a, v)\ ;\ a \in A^*,\ \mathfrak{m}^*(a) = v\}$ of memory addresses with their corresponding values. In practice, the memory is divided in several pages. Each page contains a part $\mathfrak{m}_i^*$ of the public memory, which forms a partition of the public memory:

$$\mathfrak{m}^* = \bigcup_i \mathfrak{m}_i^*.$$

The pages are part of the statement to check. The first page is a *regular* one, meaning that this page corresponds to a list of pairs address-value. The other pages are *continuous* ones, *i.e.* all the addresses in a same page are adjacent (a continuous page can be encoded by its starting address and the list of consecutive values).

2.5.2 Padding of the public memory

Some cells of the execution trace are dedicated to the public memory. The number of such cells is defined as

$$S := \frac{L}{\texttt{PUBLIC\_MEMORY\_STEP}}$$

where `PUBLIC_MEMORY_STEP` is a parameter of the layout (we introduce the notation S here for our purpose). This means that the size of the public memory $|A^*|$ must be lower than (or equal to) this number. Note that the trace length can be artificially increased to satisfy this constraint thanks to the final infinite-loop instruction (see Section 2.4).

When $|A^*|$ is strictly lower than S (which occurs most of the time), the remaining cells must be filled with 0's in the (unsorted) virtual column L_1, while they are filled with pairs $(a_{\text{pad}}, \mathfrak{m}^*(a_{\text{pad}}))$ for the (sorted) virtual column L_2, where a_{pad} is a valid address of A^* different from 0. This padding is taken into account in the formula for the final ratio p^{m} which enables to check the public memory. Instead of simply having

$$\frac{z^{|A^*|}}{\prod_{a \in A^*}(z - (a + \alpha \cdot \mathfrak{m}^*(a)))},$$

for this ratio (as explained in Section 9.8 of the Cairo paper [2]), we then have:

$$\frac{z^S}{(z - (a_{\text{pad}} + \alpha \cdot \mathfrak{m}^*(a_{\text{pad}})))^{S-|A^*|} \cdot \prod_{a \in A^*}(z - (a + \alpha \cdot \mathfrak{m}^*(a)))}.$$

2.6 Public input and verified statement

The parameters of the statement are stored in an array called the *public input*. The format of this array is defined in the smart contract `CpuPublicInputOffsets` (which further inherits from `CpuPublicInputOffsetsBase` and `PageInfo`). The structure of the public input is summarized in Table 3. We note that, the cumulative products of the public memory pages are not formally part of the public input but part of the proof.

Based on this public input, the statement to be verified is the following: *There exists a valid execution trace of T steps on the Cairo machine:*

1. *with layout corresponding to* `LAYOUT_CODE`,
2. *which starts with* $(\mathtt{pc}_I, \mathtt{ap}_I, \mathtt{fp}_I := \mathtt{ap}_I)$,
3. *which finishes with* $(\mathtt{pc}_F, \mathtt{ap}_F, \mathtt{fp}_F := \mathtt{fp}_I)$,[1]
4. *for some memory* $\mathfrak{m}$ *which is continuous read-only and consistent with a public memory* $\mathfrak{m}^* := \bigcup_i \mathfrak{m}_i^*$ *matching the pages' information from the public input array (in particular the pages' sizes and hash values),*
5. *where range-check values are between* `RC_MIN` *and* `RC_MAX`,
6. *where the unused cells for public memory are filled with* $\mathfrak{m}(a_{pad}) = a_{pad}$ *in the* L_2 *column of the execution trace,*

[1] Let us stress that $(\mathtt{pc}_F, \mathtt{ap}_F, \mathtt{fp}_F)$ should correspond the last *executed* state, and not the state after the T steps (as explained in Section 2.3.

Field	Description
LOG_N_STEPS	Log (in base 2) of the number of states T
RC_MIN	Min range-check value
RC_MAX	Max range-check value
LAYOUT_CODE	The layout identifier
PROGRAM_BEGIN_ADDR	$\mathtt{pc}_I$
PROGRAM_STOP_PTR	$\mathtt{pc}_F$
EXECUTION_BEGIN_ADDR	$\mathtt{ap}_I$
EXECUTION_STOP_PTR	$\mathtt{ap}_F$
<BUILTIN>_BEGIN_ADDR	Builtin pointer (argument of `main`)
<BUILTIN>_STOP_PTR	Updated builtin pointer (returned by `main`)
PUBLIC_MEMORY_PADDING_ADDR	a_{pad}
PUBLIC_MEMORY_PADDING_VALUE	$\mathfrak{m}(a_{\text{pad}})$
N_PUBLIC_MEMORY_PAGES	Number of public memory pages
For each page i:	
First address in the page	First address of $\mathfrak{m}_i^*$ (except for $i = 0$)
Page size	Size of $\mathfrak{m}_i^*$
Page hash	Hash of $\mathfrak{m}_i^*$
For each page i:	
Cumulative product	$\prod_{a \in A_i^*}(z - (a + \alpha \cdot \mathfrak{m}_i^*(a)))$

Table 3: Structure of the public input.

7. *where the builtin segments start at the addresses* `<BUILTIN>_BEGIN_ADDR`.[2]

We note that although $\mathtt{pc}_I$ and $\mathtt{pc}_F$ are part of the public input, the current implementation of the Cairo verifier assumes (and only supports) $\mathtt{pc}_I := 1$ and $\mathtt{pc}_F := 5$ (as explained in Section 2.4).

In practice, additional constraints must be enforced by the caller of the Cairo verifier in order to ensure the global computational integrity of the Cairo program. Namely, the above list should be completed with:

8. *where the corresponding bytecode includes a function* `main`,

9. *where the arguments of* `main` *match the set of* `<BUILTIN>_BEGIN_ADDR`,

[2]This item fixes all the addresses of the builtin segments since those segments are continuous and their size is determined by the max number of instances (defined by the builtin ratio). We stress that the validity of the execution trace requires the validity of all the instances on the builtin segment.

10. *where the returned values by* `main` *match the set of* `<BUILTIN>_STOP_PTR` *(and in particular* `main` *properly returns),*

11. *where* $\mathfrak{m}(fp_I - 2) = fp_I$ *and* $\mathfrak{m}(fp_I - 1) = 0$.[3]

These items should be verified by the caller of the `CpuVerifier` contract (that is the SHARP verifier in the context of the present audit – see Section 4), hence they should be reformulated as constraints on the public memory $\mathfrak{m}^*$.

2.7 Verifier dependencies

The CPU verifier contract is derived from several smart contracts. The parent contract `LayoutSpecific` and its own parent contract `StarkParameters` do not have a unique implementation and both depend on the selected layout. The verifier also calls external smart contracts, namely the `CpuOODS` and `CpuConstraintPoly` contracts. In practice, those four contracts are autogenerated according to the selected layout. In this section, we give an overview of those contracts.

2.7.1 Contract `StarkParameters`

To be functional, the CPU verifier requires some constants related to the format of the execution trace and the AIR constraints which are defined in the smart contract `StarkParameters` (an autogenerated parent contract of `LayoutSpecific`). We summarize those constants hereafter (the notations are from the ethSTARK documentation [3] and our previous audit [1]):

- `N_COEFFICIENTS`: Number of coefficients $\{\alpha_j\}_j \cup \{\beta_j\}_j$ to build the composition polynomial from the AIR polynomial constraints.
- `N_INTERACTION_ELEMENTS`: Number of $\mathbb{F}_p$ elements that the IOP verifier must sample for the interaction step.
- `MASK_SIZE`: Mask size M_1, *i.e.* size of $\{y_\ell\}_\ell$.
- `N_ROWS_IN_MASK`: Number of rows involved in the mask $\{y_\ell\}_\ell$.
- `N_COLUMNS_IN_MASK`: Number of columns involved in the mask $\{y_\ell\}_\ell$, which must be equal to the number of columns in the execution trace.
- `N_COLUMNS_IN_TRACE0`: Number of committed trace columns in the first round.
- `N_COLUMNS_IN_TRACE1`: Number of committed trace columns in the second round (after sampling the interaction elements).

[3] This item is required to guarantee the "safe call" feature (that is, all "call" instructions will return, even if the called function is malicious). This feature guarantees the impossibility of creating a cycle in the call stack.

- `CONSTRAINTS_DEGREE_BOUND`: Number M_2 of composition polynomial trace columns. All the constraints must have a degree of at most M_2.
- `N_OODS_VALUES`: Number $M_1 + M_2$ of OODS values $\{y_\ell\}_\ell \cup \{\hat{y}_i\}_i$.
- `N_OODS_COEFFICIENTS`: Number $M_1 + M_2$ of OODS coefficients $\{\gamma_i\}_i$.
- `PUBLIC_MEMORY_STEP`: Frequency of an address-value pair in the public memory in the execution trace. This constant defines

$$\texttt{PUBLIC\_MEMORY\_SIZE} = \frac{L}{\texttt{PUBLIC\_MEMORY\_STEP}}$$

  where L is the trace length and `PUBLIC_MEMORY_SIZE` is the size that is allocated to the public memory.
- `LAYOUT_CODE`: Unique layout identifier. Used to verify if the program to check has been compiled with the verifier's layout.
- `LOG_CPU_COMPONENT_HEIGHT`: The logarithm of the component height (see Section 2.3), satisfying

$$L = T \cdot 2^{\texttt{LOG\_CPU\_COMPONENT\_HEIGHT}}.$$

Constants about the builtins' configuration are further defined, such as `<BUILTIN>_RATIO` which corresponds to the builtin ratio (see Section 2.2), and `<BUILTIN>_REPETITIONS` for periodic columns.

2.7.2 Contract LayoutSpecific

The contract `LayoutSpecific` gathers all the layout-specific functions and in particular functions dealing with builtins. `LayoutSpecific` must implement the function `getLayoutInfo` requested by the interface `CairoVerifierContract`. To be compatible with the audited smart contract `CpuVerifier` (which inherits from `LayoutSpecific`), it must further implement the following functions:

- `initPeriodicColumns` which initializes the contracts relative to periodic columns,
- `layoutSpecificInit` which initializes the builtin-relative fields of the verifier state and which performs some sanity checks on the builtin pointers,
- `prepareForOodsCheck` which prepares the builtin-relative values for the OODS contract.

2.7.3 Contract CpuConstraintPoly

Using the notations of [3] and [1], an AIR polynomial constraint (represented as a rational function) is denoted C_j and its degree D_j. Let us recall that the composition polynomial h is then in the form

$$h(x) = \sum_{j=1}^{k} C_j(x) \left(\alpha_j x^{D-D_j-1} + \beta_j\right)$$

where k is the number of constrains, D is the smallest power of two which is greater than all the constraint degrees, and $\{\alpha_j\}_j \cup \{\beta_j\}_j$ are verifier challenges (sampled using the Fiat-Shamir heuristic while building the non-interactive proof).

Given the OODS point z, the coefficients $\{\alpha_j\}_j \cup \{\beta_j\}_j$ and some constraint-relative information (like the offsets of some boundary constraints and the interaction elements), the contract `CpuConstraintPoly` must return the evaluation of the composition polynomial at the OODS point, *i.e.* it must return $h(z)$ evaluated through the above equation.

In practice, this contract is autogenerated with the corresponding composition polynomial which is hardcoded. It uses a strategy of batching for the inverses (the denominators of rational functions C_j).

2.7.4 Contract `CpuOODS`

This contract corresponds to the OODS contract described in Section 2.5.2 of the previous audit report [1]. This contract is called by the STARK verifier while verifying the proof (specifically it is called by the auxiliary function `computeFirstFriLayer` of the `verifyProof` function). It must return the evaluations for the FRI queries of the DEEP composition polynomial and the inverses of all the evaluation points. We refer to the previous report for detailed specifications [1].

3 Review of the Cairo verifier

3.1 Management of the public inputs

3.1.1 Contract PageInfo

The contract `PageInfo` defines some constants (size and offsets) relative to the page information included in the public input of the program (for each page of the public memory – see Section 2.5.1). Specifically, it defines the size of a page information (3 memory cells, or $3 * 32$ bytes) as well as the offsets of each field in the page information, namely the first address of the page, the page size, and the page hash digest (as described in Section 2.6).

3.1.2 Contract CpuPublicInputOffsetsBase

The contract `CpuPublicInputOffsetsBase`, which inherits from `PageInfo`, defines some offsets for the fields of the public input (which are listed in Section 2.6). Specifically, it includes all the fields which are not relative to the builtins (*e.g.* `LOG_N_STEPS`, `LAYOUT_CODE`, ...) which are placed at the beginning of the public input, as well as the fields relative to the builtins which are common to each layout (namely the output, Pedersen and range check builtins). The contract `CpuPublicInputOffsetsBase` further defines a constant for the number of field elements per public memory entry (which equals to 2) as well as the initial and final values of the program counter $\texttt{pc}_I$ and $\texttt{pc}_F$ (which equal 1 and 5 as explained in Section 2.4).

Observation 1: Ambiguous constant name

The constant `N_WORDS_PER_PUBLIC_MEMORY_ENTRY` corresponds to the number of 256-bit words which represent a memory mapping for an address in the case of a regular page (which is 2). Up to our understanding, this constant name is not explicit and it does not correspond to a parameter that could take different values.

Recommendation: We recommend to simply remove this constant (and replace it by 2), or to rename it with a more explicit name (referring to regular pages) and to move it in the contract `PageInfo` since it is relative to pages of the public memory.

Status: Resolved (✓)
This constant has been renamed as `MEMORY_PAIR_SIZE` and it has been moved in the contract `PageInfo`. Moreover, a comment has been added to explain its purpose.

3.1.3 Contract CpuPublicInputOffsets

The contract `CpuPublicInputOffsets`, which inherits from `CpuPublicInputOffsetsBase`, defines the remaining offsets for the fields of the public input, specifically for the fields specific to the current layout.

It further defines getter functions for the offsets related to the page information fields with respect to the page index as well as a function returning the size of the public input with respect to the number of public memory pages.

● **Observation 2: Redundant code**

The offset getter functions are related to the page management and are the same for each layout.

Recommendation: We recommend to move those getter functions in the contract `PageInfo` since they relate to page management (or at least in the contract `CpuPublicInputOffsetsBase` since they are common to each layout). Those functions depend on the constant `OFFSET_PUBLIC_MEMORY` which could be obtained through a virtual function.

Status: Resolved (✓)

The offset getter functions are now in a dedicated contract `PublicMemoryOffsets` (which inherits from `PageInfo`). They all rely on the virtual auxiliary function `getPublicMemoryOffset` for which an implementation is provided in the contract `CpuVerifier`.

3.2 Fact registry for memory pages

3.2.1 Contract `MemoryPageFactRegistry`

Before running the Cairo verifier, all the pages $\mathfrak{m}_i^*$ of the public memory must be registered using the smart contract `MemoryPageFactRegistry`. This contract implements two registration functions depending on whether the memory page is regular or continuous. The function

```
registerRegularMemoryPage( uint256[] calldata memoryPairs, uint256 z,
                           uint256 alpha, uint256 prime ),
```

is used for a regular page, while the function

```
registerContinuousMemoryPage( uint256 startAddr, uint256[] memory values,
                              uint256 z, uint256 alpha, uint256 prime ).
```

is used for a continuous page.

Those functions first compute the cumulative product

$$\texttt{prod} = \prod_{a \in A_i^*} (z - (a + \alpha \cdot \mathfrak{m}^*(a))) \quad \in \mathbb{F}_p$$

where A_i^* is the domain of $\mathfrak{m}_i^*$ and the hash digest $h_{\mathfrak{m}_i^*}$ of the memory page computed as

$$h_{\mathfrak{m}_i^*} := \begin{cases} \text{Hash}(a_1, v_1, \ldots, a_n, v_n) & \text{if } \mathfrak{m}_i^* \text{ is a regular page,} \\ \text{Hash}(v_1, \ldots, v_n) & \text{otherwise,} \end{cases} \tag{1}$$

with $n := |A_i^*|$ and $\mathfrak{m}_i^*(a_j) = v_j$ for $j \in [1, n]$. Then, those functions register the following fact:

> *"There exists a memory mapping $\mathfrak{m}_i$ with n addresses for which the memory hash is $h_{\mathfrak{m}_i^*}$ and the cumulative product w.r.t. z and α is* `prod`*."*

This fact is stored as the hash digest

$$\texttt{ValidFact} := \text{Hash}(\texttt{pageType}, p, n, z, \alpha, \texttt{prod}, h_{\mathfrak{m}_i^*}, \texttt{address})$$

where

- `pageType` is `0` for a regular page and `1` for a continuous page,
- `address` is the first address if the page is continuous, and `0` otherwise.

The two functions further perform some sanity checks on their inputs before running the main computation. The function `registerContinuousMemoryPage` batches the terms of the cumulative product eight by eight for more efficiency.

The constants for the two page types (regular and continuous) are defined in the parent virtual contract `MemoryPageFactRegistryConstants`.

The reason why the definition of the memory hash does not have the same scope depending to the page type (it binds the addresses for regular pages, not for the continuous pages) comes from how the pages are used by the SHARP verifier (see Section 4.2).

Observation 3: Code structure

On one hand, the function `registerRegularMemoryPage` only performs the sanity checks on its input and then call an auxiliary function `computeFactHash` to perform the main computation, before finally registering the fact. On the other hand, the function `registerContinuousMemoryPage` does not call an auxiliary function and performs all the computation itself.

Recommendation: We recommend to have the same structure for the two functions (either both call auxiliary functions, or no one does).

Status: Unresolved (✗)
The two functions still have different structures.

Observation 4: Different check behavior

The check performed at lines 44 and 140, that the page size is under 2^{20}, does not behave the same way for a regular page and for a continuous page. In the former case, the actual check is that the page size is under 2^{19} since the buffer `memoryPairs` contains addresses and values (and is hence twice longer than the page).

Recommendation: Fix the above difference if not desired and document these checks.

Status: Resolved (✓)
Following some clarification from StarkWare, these checks aim to limit the input size and not the page size, thus our observation does not apply.

3.3 The Cairo verifier interface and layouts

3.3.1 Contract CairoVerifierContract

The smart contract `CairoVerifierContract` acts an interface. Its declares the external functions the Cairo verifier must implement:

- `verifyProofExternal(proofParams, proof, publicInput)` is the function which verifies the given proof with respect to the statement corresponding to the given public input.
- `getLayoutInfo()` is a function which returns some information related to the verifier layout (the offset of the public memory pages' information in the public input, the builtins selected by the layout).

The function `getLayoutInfo` returns the layout builtins as a bit-map: each builtin is associated to a bit position, and a set bit means that the corresponding builtin is selected. The contract `CairoVerifierContract` defines some constants which specify the bit position for each builtin.

● **Observation 5: Implicit interface**
As described above, `CairoVerifierContract` acts an interface even if it is not declared with the keyword `interface`.

Recommendation: We recommend to explicitly define this contract as an interface using the keyword `interface`.[a]

Status: Resolved (✓)
The use of internal constants prevents defining `CairoVerifierContract` as an interface.

[a] See `https://docs.soliditylang.org/en/v0.8.13/contracts.html#interfaces`.

3.3.2 Contract LayoutSpecific

The contract `LayoutSpecific` is an (abstract) autogenerated contract which implements some functions specific to the current layout (and to the builtins of the current layout in particular). Those functions are required by the `CpuVerifier` contract, which inherits from `LayoutSpecific`. The implemented functions are the following:

- `initPeriodicColumns` which initializes the contracts relative to periodic columns,

- `getLayoutInfo` which is declared in the interface `CairoVerifierContract` (see Section 3.3.1),
- `safeDiv` which implements a "safe" integer division (*i.e.* which verifies that the denominator is different from 0 and divides the numerator),
- `layoutSpecificInit` which initializes the builtin-relative fields of the verifier state and which performs some sanity checks on the builtin pointers. This function further calls to the subfunction `validateBuiltinPointers` which validates the consistency of builtin pointers,
- `prepareForOodsCheck` which prepares the builtin-relative values for the contract `CpuConstraintPoly` (see Section 2.7.3).

Observation 6: Unused constant

The constant `MAX_FRI_STEP` defined in the parent contract `StarkParameters` is never used. In particular, no check is performed to verify that this constant is compatible with the implementation of the STARK verifier.

Recommendation: We recommend to remove this constant, or to make proper use (or check) of it in the STARK verifier implementation.

Status: Resolved (✓)
The constant `MAX_FRI_STEP` has been removed.

Observation 7: Missing test?

Shouldn't the function `validateBuiltinPointers` further test that `stopAddress` (or `maxStopPtr`) is lower than 2^{64}?

Status: Resolved (✓)
The feedback received from StarkWare is the following: since the memory is continuous, checking `initialAddress` bounds `maxStopPtr`. We thought that the idea of the test was to check that memory addresses did not exceed 2^{64}. This check corresponds to a sanity check from a past version of the code for which the Cairo memory was not enforced to start at the address `1`.

3.4 The CPU verifier

3.4.1 Contract `CpuVerifier`

The contract `CpuVerifier` is an implementation of the interface `CairoVerifierContract`. It is also derived from the previously audited STARK verifier (contract `StarkVerifier`).

As described in Section 2.5.3 of our previous audit report [1], a contract derived from `StarkVerifier` must implement several functions. Most of them are simple getters, and `CpuVerifier` implements them by returning the values of the corresponding constants from `StarkParameters` (see Section 2.7.1). Besides those, the contract must implement three functions: `airSpecificInit`, `getPublicInputHash` and `oodsConsistencyCheck`.

The function `getPublicInputHash` returns the hash of the public input (*a.k.a.* the statement to verify). Its implementation simply hashes the public input array (defined in Table 3, Section 2.6) without the cumulative products which are not formally part of the public input.

The function `airSpecificInit` creates the verifier context and initializes the context fields which are specific to the proved statement while performing some sanity checks on the public input. The latter are described in Table 4. The function also checks:

$$\texttt{pc}_I = \texttt{INITIAL\_PC} \quad \text{and} \quad \texttt{pc}_F = \texttt{FINAL\_PC}\ ,$$

where $\texttt{pc}_I$ and $\texttt{pc}_F$ are read from the public input array and where `INITIAL_PC` and `FINAL_PC` are constants defined in the contract `CpuPublicInputOffsetsBase`, specifically `INITIAL_PC` $:=1$ and `FINAL_PC` $:=5$ in the current implementation (see Section 2.4). Finally, a call to the function `layoutSpecificInit` (see Section 3.3.2) performs some sanity checks on the initial and final builtins pointers[4], and fills the layout-related fields of the verifier context. If all the checks pass, the function returns the verifier context.

Parameter	**Check**
The number T of steps	$\log_2 T < 50$
Range-check interval	$0 \leq \texttt{RC\_MIN} \leq \texttt{RC\_MAX} < 2^{16}$
Layout Code	Must match with the verifier layout
Number of pages	At least 1, strictly lower than 100 000
Page size $\lvert A_i^* \rvert$	$\lvert A_i^* \rvert < 2^{30}$

Table 4: Sanity checks on the public input.

Finally, the function `oodsConsistencyCheck` verifies the consistency of OODS values. The implemented function proceeds as follows:

- it checks that the public memory pages have been registered as valid facts (see Section 3.2) using the auxiliary function `verifyMemoryPageFacts`. The latter computes the fact hash corresponding to each public memory page as described in Section 3.2.1 and asks the fact registry whether the computed fact hash has well been registered.
- it computes the quantity

$$\frac{z^S}{(z-(a_{\text{pad}}+\alpha\cdot\mathfrak{m}^*(a_{\text{pad}})))^{S-|A^*|}\cdot\prod_{a\in A^*}(z-(a+\alpha\cdot\mathfrak{m}^*(a)))}$$

[4]Precisely, it checks that for each builtin, the begin and end addresses of the segment verify `<BUILTIN>_BEGIN_ADDR` $\leq$ `<BUILTIN>_STOP_PTR` $\leq$ `<BUILTIN>_BEGIN_ADDR` + segment size (where the segment size is the product between the maximum number of instances and the instance size, see Section 2.2).

with `computePublicMemoryQuotient` which shall be used to check the consistency of the public memory in the execution trace (see Section 2.5). The computation of $\prod_{a \in A^*}(z - (a + \alpha \cdot \mathfrak{m}^*(a)))$ is delegated to the function `computePublicMemoryProd` which simply multiplies the cumulative products of the different memory pages from the public input array.

- it prepares the input data for the `CpuConstraintPoly` contract (see Section 2.7.3), specifically the interactive elements for public memory (z and α), the interactive element for range-check (z), the computed quotient (above item), and further builtin-relative values prepared by calling the function `prepareForOodsCheck` (see Section 3.3.2).
- it calls the contract `CpuConstraintPoly` (through a static call) which evaluates $h(z)$ where h is the composition polynomial and z is the OODS point. This value is computed from the mask $\{y_\ell\}_\ell$ using all the hardcoded AIR constraints.
- it computes $h(z)$ using the evaluations $\hat{y}_i := h_i(z^{M_2})$ of the composition polynomial trace through the equation $h(z) = \hat{y}_0 + \hat{y}_1 \cdot z$ (since $M_2 = 2$ here).
- it raises an error if the results of the two previous evaluations do not match.

■ **Observation 8: Erroneous comment**

The comment at lines 11–13 mentions "for which if a program starts at `pc`=0, it runs successfully and ends with `pc`=2" whereas the initial and final `pc` values are 1 and 5.

Recommendation: Fix the comment.

Status: Resolved (✓)
The comment now mentions the constants `INITIAL_PC` and `FINAL_PC` instead of the hardcoded values `0` and `2`.

● **Observation 9: Hardcoded restriction**

The composition polynomial $h(z)$ is defined as $h(z) = \sum_{i=0}^{M_2} z^i \hat{y}_i$ with z the OODS point and $\hat{y}_i = h_i(z^{M_2})$, where M_2 is the number of polynomial trace columns a.k.a. the degree bound on the constraints. The implementation of `oodsConsistencyCheck` assumes $M_2 = 2$ since the value $h(z)$ is directly computed as $\hat{y}_0 + z \cdot \hat{y}_1$, although a constant `CONSTRAINTS_DEGREE_BOUND` is inherited from the `StarkParameters` contract for the parameter M_2.

Recommendation: Since this restriction is an implementation choice, we recommend to check that `CONSTRAINTS_DEGREE_BOUND` is equal to 2 at the beginning of the function `oodsConsistencyCheck`.

Status: Unresolved (✗)
The feedback received from StarkWare is that static assertions do not exist in

Solidity. Even though, we would still recommend using a dynamic assertion to avoid any possibility of inconsistent use of this code with another definition of the constant `CONSTRAINTS_DEGREE_BOUND`.

● **Observation 10: Indirect offsets**

At lines 363-364, the source code gets evaluations of the composition polynomials h_0 and h_1 from the verifier context `ctx` at the offsets

`MM_OODS_VALUES + MASK_SIZE` and `MM_OODS_VALUES + MASK_SIZE + 1`.

Similarly, at line 230, it gets the padding value for public memory with

`OFFSET_PUBLIC_MEMORY_PADDING_ADDR + 1`.

Recommendation: Since there exist explicit constants for those offsets, we recommend to use them, *i.e.* to access to

- evaluations of h_0 and h_1 with `MM_COMPOSITION_OODS_VALUES`.
- padding value with `OFFSET_PUBLIC_MEMORY_PADDING_VALUE`.

Status: Resolved (✓)
The source code has been updated and the defined constants are now properly used.

3.4.2 Contract CpuFriLessVerifier

The contract `CpuFriLessVerifier` is derived from the `CpuVerifier` contract and proposes an alternative implementation of the interface `CairoVerifierContract`. It is similar to the CPU verifier (see previous section) with

- the function `verifyMerkle` (originally from the `MerkleVerifier` contract, inherited through `StarkVerifier`) overridden by the function of the same name from an instance of the `MerkleStatementVerifier` contract,
- the function `fryVerifyLayers` (originally from the `Fri` contract, inherited through `StarkVerifier`) overridden by the function of the same name from an instance of the `FriStatementVerifier` contract.

The addresses of the two contract instances are passed as additional arguments of the `CpuFriLessVerifier` constructor. Both the `MerkleStatementVerifier` contract and the `FriStatementVerifier` contract have been reviewed in our previous audit [1]. Using the `CpuFriLessVerifier` contract requires that the Merkle tree openings and the FRI transitions involved in the proof to be verified have been previously registered as valid facts using `MerkleStatementContract` and `FriStatementContract` (also previously reviewed in [1]).

3.5 General observations

Observation 11: Auxiliary functions

The following functions are auxiliary (i.e. used to avoid redundancy in the corresponding contract and/or to improve the code readability):

- `MemoryPageFactRegistry.computeFactHash`,
- `CpuVerifier.computePublicMemoryQuotient`,
- `CpuVerifier.computePublicMemoryProd`,
- `CpuVerifier.verifyMemoryPageFacts`.

Recommendation: Since these functions do not aim to be used in other contracts, we suggest to define them as private functions.

Status: Resolved (✓)
These functions are now defined as private.

Observation 12: Magic numbers

Several constant values are hardcoded without being explained or specified:

- `MemoryPageFactRegistry.sol`, lines 44 and 140: `2**20`.
- `MemoryPageFactRegistry.sol`, line 144: `2**64`.
- `CpuVerifier.sol.ref`, line 114: `2**16`.
- `CpuVerifier.sol.ref`, line 115: `2**15`.
- `CpuVerifier.sol.ref`, line 210: `0x1000000`.
- `CpuVerifier.sol.ref`: all the limits involved in the sanity checks of the public inputs (see Table 4).

Recommendation: We recommend to avoid such magic numbers by defining constants and by documenting their values.

Status: Partially resolved (∼)
Some comments have been added to document these values (for all of them except `2**16` and `2**15` in `CpuVerifier`). The comments explain that these values are somewhat arbitrary. They remain as magic numbers.

4 Documentation for the SHARP verifier

The present section provides some documentation for the audited SHARP verifier implementation which completes the available documentation [2, 3, 5].

> *SHARP (formerly known as GPS) is a service that generates proofs attesting to the validity of the executions of Cairo programs. Then, it sends those proofs to an Ethereum testnet (Goerli) where they are verified by a smart contract.*
>
> – Cairo Documentation [5]

As described above, the *shared prover* (SHARP) is a service which enables everyone to prove the computational integrity of Cairo programs. It works as follows: assume a user wants to send a proof of computational integrity for a specific Cairo program on the Ethereum blockchain, then (see Figure 3):

- The user generates a *symbolic* execution trace of its computation and sends it to the shared prover. A symbolic execution trace (called *Position Independent Execution* (PIE) trace in the Cairo runner) is an execution trace where the addresses of the different memory segments are symbolic allowing the prover to deal with memory allocation. By definition, such a trace contains the bytecode of the underlying program.
- The prover gets this execution trace and wait for more requests (from other users). A request for proving the computational integrity of a program is called a *task*.
- When the prover has received enough requests (or after a time delay), it generates a proof that all the collected execution traces are valid. This proof consists in a computational integrity proof for a program called the *bootloader* (see next section) which executes sequentially all the collected programs. To build the execution trace of the bootloader, the prover can insert the symbolic execution traces of the tasks in the bootloader's trace and deal with memory allocation of the segments.
- The prover sends this proof to an online verifier (a smart contract), called the *SHARP verifier* (formerly GPS verifier).
- This verifier checks whether the proof is valid, and if it is the case, register for each task the *fact* that the computational integrity has been verified.
- At the end, online applications can check the validity of any of those facts by calling the *fact registry contract*.

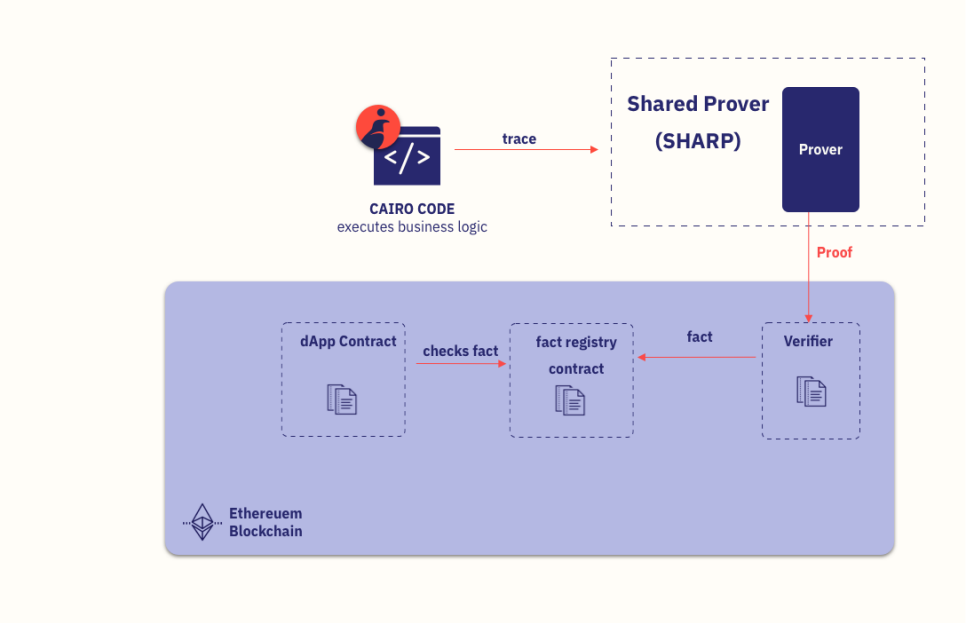

Figure 3: The SHARP environment [4].

4.1 The bootloader program

The bootloader is a special Cairo program which loads and executes a batch of programs (or *tasks*) sequentially. This approach results in a single proof for the computational integrity of all the programs. Since proof-verification scales logarithmically with the trace length, batching many computational integrity proofs in a single (larger) one enables to amortize the verification cost per task, which tends to zero while more tasks are added to the batch. The implemented bootloader uses the paradigm "bootloading from hash" (Section 2.2.1 of [2]), which enables a verifier to check the computational integrity of a Cairo program without knowing its full bytecode, but just its hash digest.

4.1.1 Simple bootloader

The program data specific to a given task is represented in Table 5. The *program header* is composed of the different fields before the program bytecode. Note that this data is not part of the public input. It is written in memory at the beginning of the task execution using a hint.

The output of the bootloader is written in a memory segment (the output builtin segment) represented on Figure 4. At the end of the execution this memory segment contains the number of tasks (first memory cell) followed by the concatenation of the output segments of all the tasks. The output segment dedicated to a given task includes a prefix, composed of the segment size and the program hash, followed by the output of the program.

To execute a given task, the bootloader proceeds as follows:

Field name	Nb. slots	Description
Data length	1	Length ℓ of the header plus the bytecode
Bootloader version	1	Version of the bootloader compliant with the program
Program main	1	Absolute position of the function `main` in the program bytecode
Number of builtins	1	Number n of builtins used by the program
List of builtins	n	List of the builtins used by the program
Program bytecode	$\ell - n - 4$	Bytecode of the program

Table 5: Program data specific to a task.

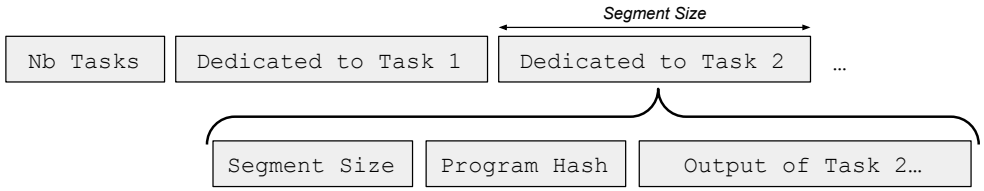

Figure 4: Bootloader output.

1. It loads the task in memory, *i.e.* writes the program's data (Table 5) in memory, which is done in a *nondeterministic way* from the first unused address.

2. it computes the hash of the program data (header and bytecode) as:

$$h = \text{Hash}(v_1, \text{Hash}(v_2, \dots \text{Hash}(v_{\ell-1}, v_\ell) \dots)) \tag{2}$$

   where $(v_1, v_2, \dots, v_\ell)$ is the memory segment described by Table 5 (with in particular $v_1 = \ell$) and where Hash is the Pedersen hash function.

3. it adds the program hash to the bootloader output using the output builtin (see Figure 4). The purpose of outputing this hash is to bind the proof to the loaded program (since the program itself is not part of the public input).

4. it computes the entry point of the program (*i.e.* address of its `main` function in the bootloader memory) and computes the builtin pointers for the program execution (as described in Section 2.4, the `main` function of a program takes the builtin pointers as implicit arguments),

5. it executes the program by jumping to its entry point.

6. Throughout its execution, the program outputs some data (using the output builtin) in a dedicated chunk (see Figure 4).

7. At the end of the task execution, the function `main` returns the updated builtin pointers which are then checked by the bootloader.

8. Finally, the bootloader writes the output segment size in the appropriate memory cell at the beginning of the segment (see Figure 4). This size must be equal to the output size of the program plus two for the segment prefix (segment size and program hash).

4.1.2 General bootloader

Instead of verifying a computational integrity (CI) statement for the simple bootloader on tasks $T_1, \ldots T_n$, an alternative approach is to consider a single task, called a *composite task*, which runs the verification of a proof π for the former CI statement. If, in addition to running the verification, such a task outputs the hash of the verified program (*i.e.* the simple bootloader) and the hash of its output (which includes the respective outputs of all the T_i's) then proving the CI statement for this composite task is equivalent to proving the former CI statement of the simple bootloader (which implies the CI of all the tasks).

The general bootloader extends the simple bootloader to support such composite tasks. Each composite task corresponds to the execution of a *Cairo verifier program* which

1. takes a STARK proof as input,
2. verifies the correctness of the proof,
3. outputs the following:
   (a) the hash of the program that is verified in the STARK proof (in practice the simple bootloader),
   (b) the hash of the output of this program (in practice a hash of the outputs of the tasks executed by the simple bootloader).

The principle of composite tasks can be applied in a recursive way: one or several of the T_i's in a composite task might be composite tasks themselves (*i.e.* tasks which execute the above Cairo verifier program on a CI proof corresponding to further subtasks). Thus, we have two types of tasks:

- *plain* tasks for arbitrary programs (such as the tasks in the input of the simple bootloader), and
- *composite* tasks which correspond to the execution of the above Cairo verifier program, and hence verifying the CI statements of several subtasks (which can be plain or composite themselves).

Thus, the full set of verified tasks can be represented as a tree where the internal nodes correspond to composite tasks while the leaves correspond to plain tasks (see Figure 5 for an illustration).

The general bootloader first calls the simple bootloader which executes the direct subtasks (*i.e.* the tasks of depth 1 in the tree, just below the root). Direct subtasks might include composite tasks whose outputs then *pack* the outputs of the underlying leaves. The general bootloader then parses the packed outputs from which it derives its own output.

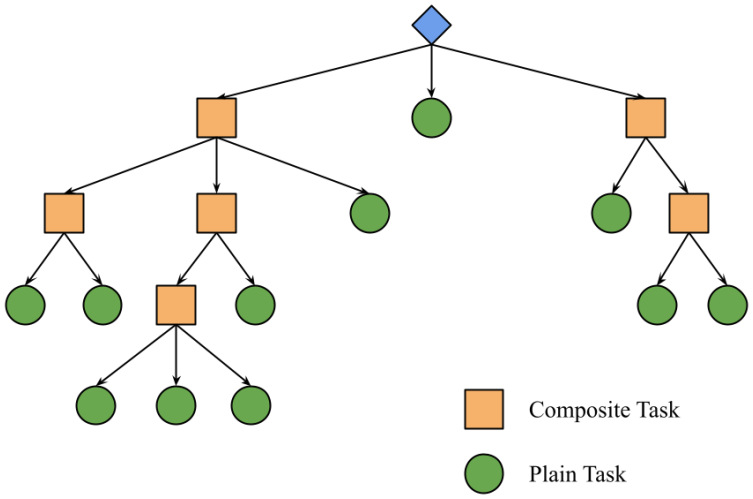

Figure 5: Tree representation of recursive tasks.

The output format of the general bootloader is given by the Figure 6. It is similar to the output of the simple bootloader (see Figure 4) with an additional configuration header. The configuration header is composed of:

- the hash of the simple bootloader bytecode which is used to execute the tasks (used to verify the first output of the Cairo verifier program used in composite tasks),
- the list of hashes of the supported Cairo verifier bytecodes (used to verify the compliance of the bytecode hash included in the output header of each composite task).

The order of the plain tasks in the output corresponds to the order of the leaves in the task tree using a depth-first search.

Figure 6: General bootloader output.

In order to unpack the tasks' outputs and reorganize them according to the format defined in Figure 6, a recursive parsing function is called on the outputs of the (plain and composite) tasks of depth 1 in the tree. For a composite task, the outputs of the corresponding subtasks are non-deterministically guessed and checked to match the second output of the Cairo verifier program. The parsing function makes recursive calls to unpack all the tasks' outputs with a depth-first search approach.

To be compatible with the audited general bootloader, the Cairo verifier program must comply with the following API:

- the input of the program is a STARK proof for the execution of the simple bootloader program (whose bytecode is part of the public input and whose output follows the format of Figure 4),
- the output of the program is the pair

$$(\text{Hash}_c(\texttt{bytecode}_{\text{sbl}}), \text{Hash}_c(\texttt{output}_{\text{sbl}}))$$

where $\texttt{bytecode}_{\text{sbl}}$ and $\texttt{output}_{\text{sbl}}$ respectively denote the bytecode and output of the simple bootloader in the STARK proof and where Hash_c is the function defined as:

$$\text{Hash}_c(v_1, \ldots, v_\ell) := \text{Hash}(\text{Hash}(\ldots \text{Hash}(\text{Hash}(0, v_1), v_2) \ldots, v_\ell), \ell)$$

with Hash being the Pedersen hash function.

4.2 Public memory for SHARP

As explained in Section 2.5, the public memory is composed of several pages. In the context of SHARP, the public memory of the proven Cairo program (the bootloader) is partitioned as follows:

- The first page, a.k.a. the *main page*, is composed of the (public) memory used by the bootloader itself (its bytecode, its inputs, its returned values, ...) including the output segment prefix of each task (segment size and program hash).
- Then the next pages (which are continuous ones) contain the outputs of the tasks, where each task can give rise to several pages.

4.2.1 The main page

The main page is a *regular* page, *i.e.* it is represented as a list of address-value pairs (see Section 2.5.1). The hash digest of this page depends on the ordering of those pairs. In what follows, we describe the content of the main page, namely the list of address-value pairs composing the page, in the same order as for the hashing operation (see Equation (1), Section 3.2.1).

The first cells are dedicated to the bytecode of the bootloader:

$$\mathfrak{m}^*(\mathsf{pc}_I) \ldots \mathfrak{m}^*(\mathsf{pc}_I + \texttt{PROGRAM\_SIZE} - 1) = \text{Bootloader bytecode}$$

where `PROGRAM_SIZE` corresponds to the size of the bootloader bytecode. Then, two cells are relative to the frame of the bootloader main function:

$$\begin{cases} \mathfrak{m}^*(\mathsf{fp}_I - 2) = \mathsf{fp}_I \\ \mathfrak{m}^*(\mathsf{fp}_I - 1) = 0 \end{cases}$$

where $\mathsf{fp}_I := \mathsf{ap}_I$. The next cells hold the arguments of the bootloader's main function:

$$\mathfrak{m}^*(\mathsf{fp}_I) \dots \mathfrak{m}^*(\mathsf{fp}_I + \texttt{N\_MAIN\_ARGS} - 1) = \text{Arguments of } \texttt{main}$$

where `N_MAIN_ARGS` is the number of arguments. Let us recall that those arguments correspond to the builtin pointers used by the bootloader (see Section 2.4). Then, come the cells for the returned values of the main function:

$$\mathfrak{m}^*(\mathsf{ap}_F - \texttt{N\_MAIN\_RETURN\_VALUES}) \dots \mathfrak{m}^*(\mathsf{ap}_F - 1) = \text{Returned values of } \texttt{main}$$

where `N_MAIN_RETURN_VALUES` is the number of returned values. In practice, those values are the builtin pointers after the bootloader execution. Finally, the last cells of the main page contain information relative to the tasks. If we denote `outputAddress` the first address of the memory segment of the output builtin, we have

$$\mathfrak{m}^*(\texttt{outputAddress}) = \text{Number of tasks}$$

and for each task T_i, we have

$$\begin{cases} \mathfrak{m}^*(\texttt{outputAddress}_{T_i}) = \text{Size of } T_i\text{'s output} \\ \mathfrak{m}^*(\texttt{outputAddress}_{T_i} + 1) = \text{Program hash for } T_i \end{cases}$$

where $\texttt{outputAddress}_{T_i}$ points to the first cell dedicated to the task T_i in the output memory segment. According to the structure described in Figure 4, we have

$$\texttt{outputAddress}_{T_{i+1}} = \texttt{outputAddress}_{T_i} + \mathfrak{m}^*(\texttt{outputAddress}_{T_i}).$$

The main page is reconstructed by the SHARP verifier in order to check the consistency of its hash and cumulative product, and register the corresponding memory fact (by calling the function `registerRegularMemoryPage`, see Section 3.2.1). This is done by the function `registerPublicMemoryMainPage` of the `GpsStatementVerifier` contract (see Section 5.1.2) using:

- the bootloader bytecode (and its size `PROGRAM_SIZE`) which is provided by an external instance of the `CairoBootloaderProgram` smart contract,
- the value $\mathsf{pc}_I :=$ `INITIAL_PC` which is hardcoded in `CpuPublicInputOffsetsBase` (parent contract of the verifier, see Section 3.1.2),
- the value `N_MAIN_ARGS` = `N_MAIN_RETURN_VALUES` = `N_BUILTINS` which is hardcoded in the `GpsStatementVerifier` contract,
- the following values from the public input (see Table 3, Section 2.6):
  - the initial frame pointer $\mathsf{fp}_I := \mathsf{ap}_I$,
  - the final allocation pointer ap_F,
  - the arguments of `main` (fields `<BUILTIN>_BEGIN_ADDR` in Table 3),

  - the returned values of `main` (fields `<BUILTIN>_STOP_ADDR` in Table 3),
  - the output address `outputAddress` (field `<BUILTIN>_BEGIN_ADDR` for the output builtin in Table 3),

- the following values from the input tasks' metadata (see Section 4.4.3 below):
  - the number of tasks,
  - the sizes of the tasks' outputs,
  - the program hashes of the tasks.

While constructing the cells for the arguments and returned values of the `main` function for a given builtin, the verifier checks whether the builtin is supported by the current layout (through the `getLayoutInfo` function, see Section 3.3.2). If the builtin is not supported, then the corresponding memory cells are set to 0. For instance, if the first builtin is not supported, we have $\mathsf{m}^*(\mathsf{fp}_I) = 0$ and $\mathsf{m}^*(\mathsf{ap}_F - \texttt{N\_MAIN\_RETURN\_VALUES}) = 0$.

4.2.2 The tasks' pages

The remaining memory pages which need to be checked compose the outputs of the different tasks. For a given task T_i, it would be possible to create a unique page which contains its full output since it consists in a continuous memory segment (see Figure 4). However, the shared prover and its verifier offers more flexibility. The output of a task can be split into several chunks and then, each chunk corresponds to a page of the public memory. This split has no impact on the prover and the verifier, but it enables a user to check only a subpart of the output of a program execution without necessarily knowing the full output (see the next section for more details).

Due to implementation choices, the pages are sorted by increasing address ranges. Moreover, the union of all the pages must include all the memory segments dedicated to the different tasks (as depicted in Figure 4) without redundancy.

Let us remark that, because a client who uses the shared prover does not know in advance where the task output will be stored in the bootloader memory, the hash of a task page must not be bound to the address range of the page but only to its content. This explains why two different definitions of the page hash are used (see Section 3.2.1).

4.3 The verified statement

The public input is the same as in the case of the Cairo verifier (see Table 3 of Section 2.6). The statement verified by the SHARP verifier is the following: *There exists a valid execution trace of T steps on the Cairo machine:*

1-7. which satisfies the constraints of the Cairo verifier, numbered from 1 to 7 in Section 2.6,

8-11. where the bytecode includes a function `main` *protected by the "safe call" feature and for which the arguments and returned values match the sets of* `<BUILTIN>_BEGIN_ADDR` *and* `<BUILTIN>_STOP_PTR` *(constraints numbered from 8 to 11 in Section 2.6),*

12. using the bootloader bytecode, meaning

$$\mathfrak{m}^*(pc_I) \dots \mathfrak{m}^*(pc_I + \texttt{PROGRAM\_SIZE} - 1) = \textit{Bootloader bytecode},$$

13. where the page $\mathfrak{m}_0^*$ *of the public memory corresponds to the main page described in Section 4.2.1,*

14. where the pages $\{\mathfrak{m}_i^*\}_{i\geq 1}$ *of the public memory correspond to the tasks' pages, described in Section 4.2.2.*

4.4 Fact registration

4.4.1 Facts from the prover

Before calling the SHARP verifier, the shared prover must first register all the public memory pages relative to the tasks' outputs (*i.e.* all the pages except the main one). This shall done by calling `registerContinuousMemoryPage` from a `MemoryPageFactRegistry` contract (see Section 3.2.1) for each page.

Then depending on the version of the Cairo verifier, the shared prover might need to register additional information before running the verifier. Indeed, Cairo verifier comes in two flavors:

- the contract `CpuVerifier` (see Section 3.4.1) for which no additional information is needed,
- the contract `CpuFriLessVerifier` (see Section 3.4.2) for which the shared prover needs to register the Merkle openings and the FRI transitions involved in the proof. Those fact registrations shall be done using the contracts `MerkleStatementContract` and `FriStatementContract` (see our previous audit report [1]).

4.4.2 Facts from the verifier

While checking a proof from the shared prover, and if the proof is valid, the SHARP verifier shall register the following fact for each task T_i:

$$\text{Hash}(\texttt{program\_hash}_{T_i}, \texttt{merkle\_root\_pages}_{T_i})$$

where

- $\texttt{program\_hash}_{T_i}$ is the hash of the program corresponding to task T_i *i.e.* the hash of the program header and bytecode for this task (see Equation (2), Section 4.1).
- $\texttt{merkle\_root\_pages}_{T_i}$ is the root of a (non-binary) Merkle tree where leafs are the page hashes (as defined in Equation (1), Section 3.2.1).

This fact encodes the following statement:

> "There exists an execution trace of the program represented by $\texttt{program\_hash}_{T_i}$ which outputs the values represented by $\texttt{merkle\_root\_pages}_{T_i}$."

The values $\texttt{merkle\_root\_pages}_{T_i}$ is computed as the root of a Merkle tree with a flexible structure (called *fact topology* of T_i), which we explain hereafter.

4.4.3 Merkle tree for pages

In the Merkle tree to compute $\texttt{merkle\_root\_pages}_{T_i}$, each node consists of a pair (h, e), where h is the hash of the child nodes (or the hash of a memory page for a leaf node) and e is the offset between the beginning of the task memory segment and the end of the (contiguous) pages represented by the node.[5] A parent node (h^*, e^*) can have have several children $(h_1, e_1), \ldots, (h_n, e_n)$, with $e_1 < e_2 < \cdots < e_n$ and is then defined as

$$h^* = \text{Hash}((h_1, e_1), \ldots, (h_n, e_n)) \quad \text{and} \quad e^* = e_n \ . \tag{3}$$

The Merkle tree structure is represented by a list of pairs which describes how to build the tree root from the leaves using a stack. This process works as follows:

1. At the beginning, we have a empty stack and the list $\{(h_i, e_i)\}$ of all pages' hashes and offsets.
2. For each pair (n_1, n_2) of non-negative integers in the Merkle tree structure,
   - push n_1 pairs (h_i, e_i) from the input list in the stack,
   - pop n_2 elements from the stack, compute the corresponding parent node as in Equation (3), and push the result in the stack.
3. At the end, all the pairs (h_i, e_i) must have been used and the stack must contain a single pair whose hash is the Merkle root.

To illustrate this process, Figure 7 shows the successive states of the stack when running the above algorithm on the list of pages' hash digests $[h_1, h_2, h_3]$ and with the Merkle tree represented as `[(2,2),(1,2)]`. In this figure, we have $h_{1,2} := \text{Hash}((h_1, e_1), (h_2, e_2))$ and $h_{(1,2),3} := \text{Hash}((h_{1,2}, e_2), (h_3, e_3))$. The resulting Merkle tree is further represented in Figure 8.

The structures of the Merkle trees used to compute the facts for the different tasks are provided as input of the SHARP verifier in the so-called *tasks' metadata*. The latter is an array of format depicted in Table 6.

[5] In other terms, the address range represented by the node (*i.e.* by all the memory pages corresponding to the leaves of this node) is $\{a + x, \ldots, a + e - 1\}$ where a is the first address of the memory segment dedicated to the current task and x is a non-negative integer smaller than e.

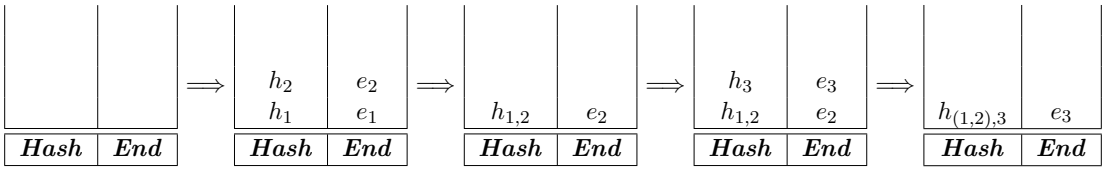

Figure 7: Stack transitions to build the Merkle root from the list `[(2,2),(1,2)]`.

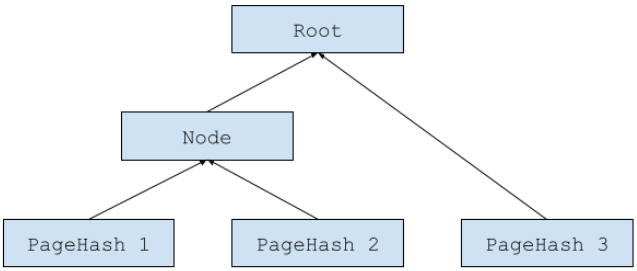

Figure 8: Example of a fact topology for a task.

Number of tasks (n)
Size of T_1's output (sum of pages' sizes)
Program hash for T_1 ($\texttt{program\_hash}_{T_1}$)
Number of pairs in the Merkle tree of T_1
Merkle tree structure (list of pairs) for T_1
Size of T_2's output (sum of pages' sizes)
Program hash for T_2 ($\texttt{program\_hash}_{T_2}$)
Number of pairs in the Merkle tree of T_2
Merkle tree structure (list of pairs) for T_2
$\vdots$
Size of T_n's output (sum of pages' sizes)
Program hash for T_n ($\texttt{program\_hash}_{T_n}$)
Number of pairs in the Merkle tree of T_n
Merkle tree structure (list of pairs) for T_n

Table 6: Tasks' metadata

5 Review of the SHARP verifier

5.1 SHARP verifier contracts

5.1.1 Contract `GpsOutputParser`

The contract `GpsOutputParser` implements the function `registerGpsFacts` which aims to register for each task the fact that the computational integrity has been checked. To proceed, for each task T_i, this function

- computes the hash digest $\texttt{merkle\_root\_pages}_{T_i}$ which is the root of a non-binary Merkle tree taking the memory pages' hashes as input (see Section 4.4) ,
- computes the fact hash of the task defined as

$$\texttt{fact\_hash}_{T_i} := \text{Hash}(\texttt{program\_hash}_{T_i}, \texttt{merkle\_root\_pages}_{T_i})$$

  where the program hash digest is available from the tasks' metadata (see Table 6),
- registers $\texttt{fact\_hash}_{T_i}$ using the `registerFact` function from `FactRegistry`.

At the same time, the function checks that the memory pages of each task well form a partition of the output memory segment dedicated to this task. Namely it checks that

- the first page starts with the first address of the memory segment,
- the first address of each page is contiguous to the last address of the previous page,
- the total size of the memory pages equals the size of the memory segment.

The function also emits an event to log the Merkle root together with the memory pages' hashes of the task (input of the Merkle tree). The logged data is a buffer `pageHashesLogData` defined as

- $\texttt{pageHashesLogData}[0] = \texttt{merkle\_root\_pages}_{T_i}$
- $\texttt{pageHashesLogData}[1] = \texttt{0x40}$
- $\texttt{pageHashesLogData}[2] = \ell$ (number of memory pages for task T_i)
- $\texttt{pageHashesLogData}[3] =$ hash of page 1 of T_i

  $\vdots$
- $\texttt{pageHashesLogData}[\ell + 2] =$ hash of page ℓ of T_i

Observation 13: Unclear implementation choice

When building a Merkle node by hashing its child nodes, the resulting hash digest is incremented by one. According to the comment at line 263, adding one to the node hash enables to "distinguish it from the hash of a Merkle leaf". Since the used hash function is `keccak256` in the both cases, it is not clear why and how this increment

helps to distinguish.

Recommendation: Update the comment to provide a more detailed explanation.

Status: Resolved (✓)
A more detailed comment has been added at lines 51-52 explaning the purpose of this increment which is to enforce some domain separation, *i.e.* to have distincts hash functions for the leaves and for the nodes.

■ **Observation 14: Mismatch between comment and code**

According to the comment line 29, the event `LogMemoryPagesHashes` "logs the fact hash together with the relevant continuous memory pages' hashes". However, when emitted (lines 182–196) it logs the Merkle root (variable `factWithoutProgramHash`) together pages' hashes (as described above) but not the fact hash (variable `fact`).

Recommendation: Correct the comment or the code.

Status: Resolved (✓)
The comment for the event `LogMemoryPagesHashes` has been fixed. Moreover, the variable `factWithoutProgramHash` has been renamed as `programOutputFact`.

5.1.2 Contract `GpsStatementVerifier`

The contract `GpsStatementVerifier` implements the function `verifyProofAndRegister` which verifies a SHARP proof and registers the associated facts.

This contract uses three state variables:

- `bootloaderProgramContractAddress` which points to a smart contract storing the bytecode of the bootloader,
- `memoryPageFactRegistry` which points to an instance of `MemoryPageFactRegistry` (see Section 3.2),
- `cairoVerifierContractAddresses` an array of pointers to different instances of the contract `CairoVerifierContract` (see Section 3.3.1).

The function `verifyProofAndRegister` takes as input the proof parameters, the SHARP proof, the tasks' metadata (see Section 4.4.3), the public input (extended with the two interaction elements z and α required to check the memory – see Section 2.6), a Cairo verifier identifier. The latter identifier is used to select the desired Cairo verifier instance from the array `cairoVerifierContractAddresses`.

The function `verifyProofAndRegister` proceeds as follows:

1. First, it builds and registers the main page of the statement (as detailed in Section 4.2.1) using the auxiliary function `registerPublicMemoryMainPage`. Note that the other pages corresponding to the tasks' outputs should have been previously registered by the shared prover.
2. It verifies that the reconstructed and registered main page matches the page size, hash and cumulative product from the public input (to avoid calling the Cairo verifier in case of mismatch).
3. It calls the selected Cairo verifier instance (function `verifyProofExternal`, see Section 3.3.1) to verify the proof.
4. If the proof verification succeeds, it finally registers the fact that the computational integrity has been verified for each task, using the `registerGpsFacts` function from `GpsOutputParser` (see Section 5.1.1).

● **Observation 15: Constant definition**

The contract `GpsStatementVerifier` defines a constant for the number of builtins:

```
uint256 internal constant N_BUILTINS = 5;
```

However some other contracts exist for the purpose of defining such constants, and in particular the interface `CairoVerifierContract` (which defines the constants related to the builtins' selection bitmap).

Recommendation: Move this constant definition to `CairoVerifierContract`.

Status: Resolved (✓)

The constant `N_BUILTINS` aims to refer to the number of builtins managed by the corresponding bytecode (here, by the bootloader). This number can be larger than the number of layout builtins. We still recommend to rename this constant to avoid any confusion (for example, `N_PROGRAM_BUILTINS` or `N_BOOTLOADER_BUILTINS`).

■ **Observation 16: Confusing comment**

The comment in line 86 "Each page has a page info and a hash" to explain that the length of `publicMemoryPages` is expected to be `nPages * (PAGE_INFO_SIZE + 1)` is not accurate. The explanation for this expected value is

- each page has page info and a *cumulative product* (hash is part of the page info),
- the page info of the first page is of size `PAGE_INFO_SIZE - 1` (it does not include the page address),
- the `publicMemoryPages` array includes the number of pages as first element.

Recommendation: Update the comment to reflect the above explanation.

Status: Resolved (✓)
The comment has been updated (lines 102-105) as recommended.

5.1.3 General observations

Hereafter are some observations which are general to the two contracts of the SHARP verifier.

● **Observation 17: Auxiliary functions**

The following functions are auxiliary (i.e. used to avoid redundancy in the corresponding contract and/or to improve the code readability):

- `GpsOutputParser.pushPageToStack`,
- `GpsOutputParser.constructNode`,
- `GpsStatementVerifier.registerPublicMemoryMainPage`.

Recommendation: Since these functions do not aim to be used in other contracts, we suggest to define them as private functions.

Status: Resolved (✓)
These functions are now defined as private.

● **Observation 18: Magic numbers**

Many constant values are hardcoded:

- `GpsOutputParser.sol`, line 123: `2**20`, the maximum number of pages by *push* operation.
- `GpsOutputParser.sol`, line 224: `2**30`, the maximum size for a page of the public memory.
- `GpsStatementVerifier.sol`, line 153: `2**30`, the maximum number of tasks.
- `GpsStatementVerifier.sol`, line 255: `2**30`, the maximum number of output slots used by a task.
- `GpsStatementVerifier.sol`: line 259: `2**20`, the maximum size of the structure which describe the fact topology of a task.

Recommendation: We recommend to avoid such magic numbers by defining constants and by documenting their values. Since the maximum size for a page is also used in the Cairo verifier, we recommend to define the corresponding constant in the contract `PageInfo`.

Status: Partially resolved (∼)
Some comments have been added to explain that these values are somewhat arbitrary. They still remain as magic numbers. Moreover, even though these values are arbitrary, we recommend to explain why the corresponding sanity checks are needed. For example, what is the interest of limiting the number of tasks?

5.2 Bootloader source code

5.2.1 File `simple_bootloader.cairo` (main function for the simple bootloader)

The entry point of the simple bootloader (see Section 4.1.1) is the function `main` from the file `simple_bootloader.cairo`. Using hints, this function parses the input files and produces some output files while executed by the Cairo runner, but from the Cairo machine point of view, it simply calls the subfunction `run_simple_bootloader`.

5.2.2 File `bootloader.cairo` (main function for the general bootloader)

The entry point of the general bootloader is the function `main` from the file `bootloader.cairo`. This function starts by calling the function `run_simple_bootloader` to execute the simple bootloader on the direct subtasks (see Section 4.1.2). However, instead of passing the real output builtin pointer as an (implicit) argument to this call, it passes a pointer to another memory area. After the execution of `run_simple_bootloader`, this memory area contains the output of the simple bootloader. For the rest of its computation, the general bootloader parses and reorganizes this output as explained in Section 4.1.2.

The function first writes the bootloader configuration at the beginning of its own output segment (see Figure 6) using the subfunction `serialize_bootloader_config`. The latter simply writes:

- the hash digest of the simple bootloader bytecode verified by the Cairo verifier program in composite tasks, and
- the hash digest of the list of the possible bytecodes for the Cairo verifier program.

Then, the function lets an empty cell in the output memory segment which will be assigned to the total number of plain tasks (this number is not known yet and will be progressively computed during the depth-first search of the task tree).

By calling the subfunction `parse_tasks`, the bootloader initializes the depth-first search of the task tree. It browses recursively each branch of the tree. When it finds a plain task (a leave of the task tree), it calls the function `unpack_plain_packed_task` which

simply copies the chunk dedicated to this task from the simple bootloader output (see Figure 4) to the general bootloader output (see Figure 6) and which increments the counter of the number of plain tasks. When it encounters a composite task, it calls the function `unpack_composite_packed_task`. The latter proceeds as follows:

1. It non-deterministically guesses the output of the simple bootloader, *i.e.* the pre-image of the hash digest output by the Cairo verifier program.
2. It verifies that the guessing is right by hashing the output and checking the hash digest.
3. It verifies that the hash digest of the Cairo verifier program is in the list of the supported program hash digests from the general bootloader configuration.
4. It verifies that the hash digest of the simple bootloader output by the Cairo verifier program matches the one of the general bootloader configuration.
5. It calls `parse_tasks` on the guessed output to recursively browse the subtasks of the current composite task.

5.2.3 File `run_simple_bootloader.cairo` (main loop)

The function `run_simple_bootloader` initializes some variables and implements the main loop which executes the different tasks. Specifically, this function

- writes the number of tasks to the output segment,
- initializes the lists `builtin_encodings` and `builtin_instances_sizes` that shall be given as input of the core function,
- initializes a range-check pointer `self_range_check_ptr` which will be dedicated to the verification of the builtin pointers returned by the task programs. This pointer gives access to a memory segment of length `n_tasks` × `n_supported_builtins` where `n_tasks` is the number of tasks and `n_supported_builtins` is the number of builtins the bootloader supports (currently five).
- initializes a list of builtin pointers which will be used by the task programs. In this list, the range-check pointer points to the end of the previous range-check segment (the segment pointed by `self_range_check_ptr`),
- calls the auxiliary function `execute_tasks` which calls the function `execute_task` (from `execute_task.cairo`) on each task,
- checks that the memory segment initially pointed by `self_range_check_ptr` has been completely consumed (*i.e.* after execution of the tasks `self_range_check_ptr` matches the initial range-check builtin pointer given to task programs),

- checks that the range-check builtin pointer has been effectively advanced beyond the new value of `self_range_check_ptr` (this is done by checking that the difference between the two pointers is non-negative).

Observation 19: Check of the RC builtin pointer

Before returning, the function `run_simple_bootloader` checks that the range-check builtin pointer has been effectively advanced by calling `verify_non_negative`. However the aim of this check is not clear since the function `execute_task` (from the file `execute_task.cairo`) already checks that the builtin pointers used by the task programs have correctly advanced (see next section). Moreover, it is not clear why this builtin pointer has a special treatment compared to the other builtin pointers used by task programs.

Recommendation: Document the purpose of this check.

Status: Resolved (✓)

A comment was added at lines 94–102 of `run_simple_bootloader`. This refers to the "Cairo calling convention", a convention for Cairo developers stating that only the builtin instances between the initial and the final values of a builtin pointer are assumed to be verified by the proof system. Without the check of the range-check builtin pointer, the function `run_simple_bootloader` could return the same range-check builtin pointer that it received as input. Indeed, even if `execute_tasks` is called with this pointer trustfully advanced at the end of the self range-check segment, the latter function calls untrusted code (the tasks' bytecode), so that the range-check pointer returned by `execute_tasks` (and then by `run_simple_bootloader`) might have stepped back. In theory, this malicious behavior would be prevented by `validate_builtins`, but the verification performed by this function is only valid if the instances of the self range-check segment are verified, which might not be the case –according to the aforementioned calling convention– if the range-check builtin pointer returned by `run_simple_bootloader` have stepped back (before the end of the self range-check segment).

It seems to us that this check is not necessary strictly speaking while considering how the builtin instances are currently verified in SHARP. In practice, the AIR constraints verify all the instances in the builtin segment (not only the instances between the initial and final values of the builtin pointer). The above malicious strategy would require that the self range-check segment (which is at the beginning of the range-check segment) exceed the range-check segment, which seems unlikely (except if the range-check builtin ratio is particularly large or if the trace is particularly small) and which would be detectable from the statement (because the statement contains the Cairo layout identifier and the number of tasks as part of the output).

Nonetheless, having this check of the RC builtin pointer enforces the respect of the "Cairo calling convention" and results in a bootloader bytecode which is secure

regardless of how the builtin instances are verified. This is safer and we are now convinced of the usefulness of this check.

5.2.4 File `execute_task.cairo` (task execution)

The core of the bootloader program resides in the `execute_task` function from the file `execute_task.cairo`, which executes a single task as described in Section 4.1. The task to be executed is passed via the hint variable `task`. The `execute_task` function further takes as input:

- The list `builtin_encodings` of builtins that the bootloader supports. Each builtin is represented by a field element, called *encoding* of the builtin. This list *must* be sorted in the same order as the builtins' pointers (implicit arguments of Cairo programs).
- The list `builtin_instance_sizes` which associates, for each builtin, the size of the builtin instance.
- As implicit argument, a list `builtin_ptrs` which for each builtin gives a pointer to the first unused instance.
- As implicit argument, a pointer `self_range_check_ptr` which is another pointer for the range-check builtin. The function hence gets two different pointers for the range-check builtins. Those two pointers point to two distinct areas of the range-check builtin memory segment. The program needs two pointers to this segment because, for each task, after running the task program, the bootloader shall check that the updated builtin pointers are valid (and in particular the range-check builtin pointer), and for this purpose the bootloader needs to use another part of the range-check builtin segment, namely the part pointed by `self_range_check_ptr`. The correct usage of the latter part of the range-check builtin segment is checked by the `run_simple_bootloader` function, once all the tasks have been executed (see Section 5.2.3).

In practice, the two non-implicit arguments are constants and, in the audited version of the bootloader program, are equal to:

builtin_encodings	builtin_instance_sizes
'output'	1
'pedersen'	3
'range_check'	1
'ecdsa'	2
'bitwise'	5

Let us recall that `'xxxxx'` is a short string literal which is encoded as a single field element.

First of all, the function `execute_task` gets the address `program_data_header` of the task headers and bytecode. This address is given via a hint, and the memory area pointed

by the address is also filled with a hint. The function then checks that the loaded program is compatible with the current bootloader version thanks to the corresponding header. Let us note that at this point nothing ensures that the right bytecode was loaded.

To convince the verifier that the loaded program is the right one, the function then computes the hash digest of the corresponding memory segment as described in Equation 2 (see Section 4.1) and writes the result in the output segment (using the output builtin). This hash computation is done by the auxiliary function `hash_chain` which makes use of the Pedersen builtin (through the pointer `builtin_ptrs.pedersen`). Using a hint, the function further recomputes the expected hash digest from the program of the `task` hint variable and verifies that it matches the digest written in output.

The function then computes the entry point of the program as

```
program_entry_point = program_address + program_main
```

where `program_address` is the program's address stored after the header in the bootloader memory and `program_main` is the absolute position (available from the program header) of the function `main` of the loaded program (see Table 5, Section 4.1). After computing the entry point, the function prepares the builtin pointers which will be given as input to the task program:

- it increases the output builtin pointer by 2, because the bootloader uses the two first slots for the output prefix which is composed the segment size and program hash (see Figure 4),
- it sets the Pedersen builtin pointer to the value returned by the `hash_chain` function (which used some Pedersen slots to compute the program hash),
- the other builtin pointers are left unchanged.

Those updated pointers are stored in the list `pre_execution_builtin_ptrs`. This list contains the pointers for all the five builtins. However the task program might not use all of them, but only a subset of them (*i.e.* the `main` function of the program might take less than 5 builtin pointers as implicit arguments). That is why the bootloader needs to select the appropriate builtin pointers according to the list `builtin_list` from the program header. The selection is done by calling the auxiliary function `select_input_builtins` which returns the pointers of the selected builtins (at [ap-1], [ap-2], ...). This process is illustrated in Figure 9 for an example program which only uses the output and range-check builtins, where the `select_input_builtins` function shall store $\texttt{ptr}_{\text{output}}$ in [ap-2] and $\texttt{ptr}_{\text{rc}}$ in [ap-1].

At this point, the function calls the program with an absolute call to the entry point:

```
call abs program_entry_point.
```

From the Cairo runner point of view, this call can be done in two different ways (depending on the task data processed by the surrounding hints):

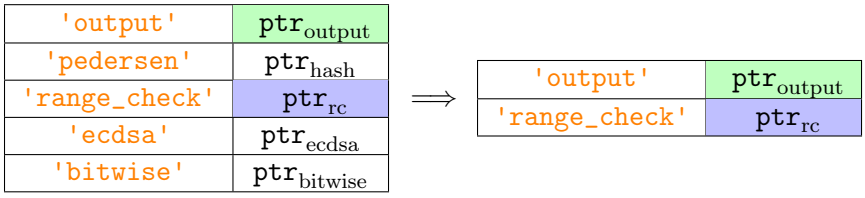

Figure 9: Selection of builtin pointers.

- either the Cairo runner has already the symbolic execution trace (a.k.a. the PIE trace) that the bytecode execution produces and then simply append this trace to the bootloader trace (also dealing with memory segment allocation) rather than re-executing the task bytecode,
- or the Cairo runner does not have such symbolic execution trace and then has to run the bytecode itself.

When the task program returns, the updated builtin pointers are available in memory at [ap-1], [ap-2], ... The idea is now to compute the list `return_builtin_ptrs` of all the five builtin pointers by updating the pointers of the selected builtins with the returned builtin pointers from the task execution. This is illustrated on Figure 10 using the same example as above. In practice, the list `return_builtin_ptrs` is computed thanks to a hint, and a call to the auxiliary function `inner_select_builtins` checks that this list is really a extension of the pointers returned by the task program. This call makes use of the list of selected builtins `builtin_list` from the program header and returns a pointer to the end of this list. The function then checks that this pointer has well been incremented of `n_builtins`, the number of builtins from the program header.

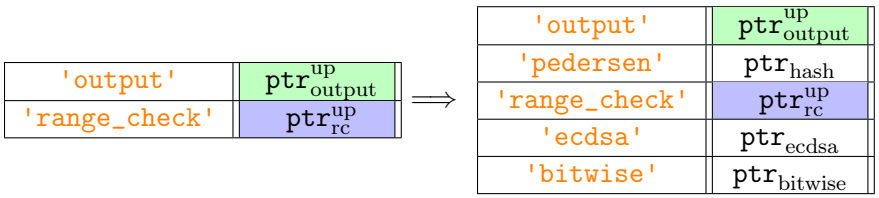

Figure 10: Returned builtin pointers.

Next, the function `validate_builtins` compares the updated list of builtin pointers (namely `return_builtin_ptrs`) to its original value before running the program (namely `pre_execution_builtins_ptrs`). For each builtin, the function `validate_builtins` checks that the corresponding pointer has increased by a multiple of the builtin instance size and that the number of used builtin instances is between 0 and 2^{128}, the range-check bound.[6] Note that this process does not strictly ensure that the builtin pointers which are not used

[6]The upper bound depends on the configuration of the range-check builtin via the parameter `RC_N_PARTS`. This check fixes an implicit maximum for the number of used instances. For example, it means that the number of outputs of the task program is limited. But in practice, the maximum if very high constraint is very wide, so the limitation is not an issue.

by the task program are the same between both lists. A malicious prover could indeed advance the builtin pointers which are not used by the task program. However, while doing so would waste some builtin memory space, it would not allow to prove a false CI statement since the Cairo verifier checks that the builtin pointers returned by the main function are valid (*i.e.* are before the end of the respective builtin memory segments).

Finally, the function `execute_task` puts in the output segment the number of output slots consumed by the task program (see Figure 4), use some hint to store the fact topology of the executed task and set the implicit return value of `builtin_ptrs` to the address of the new list `&return_builtins_ptrs`.

■ **Observation 20: Absolute positions in the bytecode**

According to the user documentation [5], the instruction `call` with an absolute position is not supported by Cairo although it is defined in the Cairo article [2] (and used by the bootloader). Presumably, this restriction aims to ensure that the bytecode is position-independent and can hence correctly be used by the bootloader (which deals with memory allocation for tasks). On the other hand, the jump at absolute position (`jmp abs` instruction) seems to be supported. Why this difference of treatment?

Recommendation: Document the support (or non-support) of absolute positions in the bytecode.

Status: Resolved (✓)

The user documentation [5] has been updated and now states that the instruction `call` with an absolute position is supported:

> The full syntax of `call` is similar to `jmp`: you can call a label (a function is also considered a label), and make a relative or absolute call (`call rel/abs ...`).
>
> – "Functions" page.

References

[1] CryptoExperts (Thibauld Feneuil and Matthieu Rivain). Code Review of StarkWare's EVM STARK Verifier, December 2021.

[2] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. https://ia.cr/2021/1063.

[3] StarkWare. ethstark documentation. Cryptology ePrint Archive, Report 2021/582, 2021. https://ia.cr/2021/582.

[4] StarkWare. Cairo for Blockchain Developers, 2022. https://www.cairo-lang.org/cairo-for-blockchain-developers/.

[5] StarkWare. StarkNet and Cairo Documentation, 2022. https://www.cairo-lang.org/docs/.