PUBLIC

Limited Code Review

of the Sway Optimizations

June 24, 2024

Produced for

FUEL

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Sway Team,

Thank you for trusting us to help you with this limited code review. Our executive summary provides an overview of subjects covered in our review of the latest version of Sway Optimizations according to the Scope.

Limited code reviews are best-effort checks and do not provide assurance comparable to a non-limited code assessment. This review was not conducted as an exhaustive search for bugs, but rather as a best effort sanity check. Given the large scope and codebase and the limited time, the findings are not exhaustive.

Fuel implements various optimization passes for the IR. These aim to facilitate the bytecode generation that follows in the later steps of the compilation, as well as to improve the overall efficiency of the compiled program, both in terms of size and execution cost.

During the review we were able to uncover a medium severity issue regarding function deduplication. More specifically, functions that are not functionally identical could be assumed as such. As a result, calls to some of them would be replaced with calls to another one.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	2
• Code Corrected	1
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The review consisted of a non-exhaustive general review of the following IR optimizations under `sway-ir/src/optimize/`:

- `constants.rs`
- `dce.rs`
- `inline.rs`
- `fn_dedup.rs`
- `simplify_cfg.rs`
- `arg_demotion.rs`
- `ret_demotion.rs`
- `const_demotion.rs`
- `mem2reg.rs`

This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	6 May 2024	8d50370814b62cff5412c6ca896efb731da49193	Initial Version
2	13 May 2024	8c999fa2520a7ee49e270d6469647d814ebc1347	Print IR optimizations
3	24 June 2024	e77855811978811ca3a74839a6798a62bdfadd82	Division by 0 fix
4	24 June 2024	36b6b010eb2ab2318c5d4d17ffc3df3e593fe60b	Fndedup fix

2.1.1 Excluded from scope

All other files and imports that were not mentioned in Scope. Moreover, bugs related to the Rust compiler itself are considered out-of-scope.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Fuel offers the optimizations on Sway's intermediate representation (IR). Sway is a programming language that targets the FuelVM. The compilation process can be split into the following steps:

1. Source Code: A set of scripts/predicates/libraries/contracts written in Sway and organized in modules.
2. Lexical Analysis: The source code is split into tokens.
3. Syntax Analysis: The tokens are assembled in a concrete syntax tree which is then converted to an abstract syntax tree (AST).
4. Semantic Analysis: Various checks on the AST for semantic errors e.g., type checking.
5. Intermediate Representation: The syntax tree is compiled into IR.
6. **IR Optimizations**: The IR is transformed in various ways to facilitate code generation and improve the efficiency of the final bytecode.
7. Code generation: FuelVM bytecode is produced.
8. Deployment & Execution

2.2.1 The Sway IR

The Sway IR is a representation of the source code in a single static assignment form (SSA), meaning that every variable is assigned at most once. The Sway IR defines a number of functions. Each function accepts a set of arguments, defines a set of local variables, and consists of multiple blocks. Each block accepts a set of arguments and defines a set of instructions which are executed sequentially. At the end of each block, there's a terminator instruction which guides the execution to the next block. A terminator can be a simple branch instruction (unconditional jump), a conditional jump, a return instruction, a revert instruction, or a memory jump (used for FuelVM's equivalent for delegate calls). Data can flow through different blocks via the block arguments (see example later). Most instructions produce a typed value that is assigned to an intermediate variable, exceptions are the terminators and the storage operations.

Here, we present a very simple script example of a Sway source code and its IR representation. The script consists of two functions. The `main` function accepts two arguments and the `max` function returns the maximum of its two arguments.

A Sway script example:

```
script;

fn main(a: u64, b: u64){
    let c = max(a, b);
}

fn max(l: u64, r: u64) -> u64 {
    if l > r {
    l
    } else {
        r
    }
}
```

An example of the produced IR:

```
fn main(a: u64, b: u64) -> () {
    local u64 c

    entry(a: u64, b: u64):
    v0 = call max(a, b)
    v1 = get_local ptr u64, c
    store v0 to v1
    v2 = const unit ()
    ret () v2
}


fn max(l: u64, r: u64) -> u64 {
    entry(l: u64, r: u64):
        v0 = cmp gt l r
        cbr v0, block0(), block1()

    block0():
        br block2(l)

    block1():
        br block2(r)

    block2(v1: u64):
        ret u64 v1
}
```

Examining the `max` function in the IR of the script, we note that the execution begins from the `entry()` block. The arguments `l` and `r` are compared and the result is assigned to `v0`. If `v0 == true` then the execution continues in `block0()` from where we simply jump to `block2` passing `l` as the argument. Then the argument of the block is returned.

`main` defines a local variable `c`. `c` is assigned the result of `max()` which is stored in `v0` intermediate variable. To do so we first get a pointer to `c` via `get_local` instruction which is stored in `v1` and then store the content of `v0` to `v1` by calling `store v0 to v1`.

The IR goes through multiple passes. Each optimization transforms the IR and passes it on to the next one. In the following section, we describe the algorithm of each optimization in scope.

2.2.2 Inline

The `inline` pass replaces function calls with the bodies of the functions being called. In the current implementation of the algorithm, a function is inlined if either of the following criteria holds:

1. The size of a function is less than 4 instructions.
2. The function is called only once.

During inlining the block containing the function call is split into two blocks at the point of the call. In the first block, the call is replaced by instructions of the first block of the inlined function. Each return instruction of the function is replaced by an unconditional jump to the second block. Since caller and callee might define local variables with the same name, these should be renamed to avoid conflicts.

2.2.3 Function Deduplication

The `fn-dedup` pass aims to detect functions that are functionally identical. To do so, it calculates the hash of all the functions. For the hashing to be meaningful, the globally-unique ids of values and blocks are replaced with localised ones, so that equality detection is not hindered by, say, two identical instructions having a different global id. The types of the various values are also hashed.
Functions are divided into equivalence classes based on their hash. All the calls to a (duplicated) function are then replaced with calls to a chosen representative for its equivalence class, so that the others can later be pruned out.

2.2.4 Dead Code Elimination (DCE)

The `dce` pass removes the instructions and the local variables that are never used. An instruction that is never used can be eliminated in either of the two cases:

1. The instruction is not a terminator and it has no side effects (e.g., changes the state of the blockchain or writes to memory)
2. The instruction is a removable store i.e., a store to a local variable or an argument that's never used. The algorithm is very conservative in determining what symbols are never used. For that, it takes into account the escaped symbols analysis e.g., symbols that could be used in function calls or whether raw pointers and referencing are used.

Note that as soon as we delete an instruction, its operands are likely to not be used somewhere else therefore they should be eliminated as well. The algorithm loops until a fixed point is reached.

2.2.5 Function Dead Code Elimination (FN-DCE)

During the `fn-dce` pass, internal functions that are never called are removed. Note that no other change is needed for the remaining code. The called functions are determined by a DFS starting from the possible entry functions of the module.

2.2.6 Simplify-CFG

The simplification of the control flow graph (CFG) consists of multiple different steps:

1. First dead block removal: The blocks of each function are traversed in a DFS fashion starting from the entry block. The blocks that were never accessed are removed. The successors of the removed blocks update their predecessors list.
2. Block merging: If the terminator of a block is an unconditional branch and its successor has only one predecessor then the two blocks are merged into one. The successors of the latter block are updated.
3. Empty block unlinking: an empty block that accepts no arguments and has an unconditional branch terminator is unlinked from the CFG.
4. Second dead block removal: repeat of the first step

2.2.7 Constant demotion

Some IR types are classified as "demotable", based on the target underlying platform: for the Fuel VM target, these are all the elementary types that are too wide to fit in the 64-bit registers (e.g U256), plus all the complex types (e.g. structs) regardless of their size.
The `const-demotion` pass takes every constant of a demotable type, and replaces every occurrence of it with a new, unique, and immutable local variable, initialized to that constant.

2.2.8 Argument demotion

The `arg-demotion` pass acts on function parameters of demotable types: it modifies the function so that the parameter is passed by reference, rather than by value.
To achieve this, the function signature needs to be modified, so that the parameter in question becomes of pointer type (`arg: T` becomes `arg: &T`).
Then, in the function body, a `load(arg)` instruction is prepended at the beginning, and its result replaces every usage of the old by-value argument. In high-level language terms, this corresponds to substituting `arg` with `*arg`.

Lastly, every caller needs to be updated. At every call site, the concrete argument value is stored in a new, ad-hoc local variable. Then, the pointer to that variable is passed to the function, instead of the original argument.
Notice that this optimization does not reduce the overall memory footprint: the argument value is still copied onto memory once per call (in the callee's stack frame, before the pass, in the caller's local variable, after the pass). Instead, the purpose is to exclusively have function parameters that fit in a register, so as to ease the work of the code-generation algorithms further down the line (e.g. register allocation).

2.2.9 Return-value demotion

The `ret-demotion` pass complements the previous one by demoting the return value of a function if the type allows it.
The function signature is modified so as to accept an *additional* parameter, whose type is a pointer to the return type. The return type is likewise modified to become a pointer. The storage for the return value is provided by a new ad-hoc local variable in the caller: the additional parameter is a pointer to that variable and is returned as-is by the function.
The function body is updated to simply `store` the final return value (at every return site) into the provided pointer argument, and to return the same pointer.
At every call site, the ad-hoc local variable is created, and a pointer to it is passed to the function. After the call, a `load(ret_val)` instruction is appended, and its result replaces every usage of the old return value. In high-level language terms, this corresponds to substituting `ret_val` with `*ret_val`.
Again, this optimization is not aimed at slimming down the total memory requirements, but at making function signatures deal exclusively with register types.

2.2.10 Constant Folding

The `const-folding` pass eliminates operations whose result can be determined during compilation time. The optimization consists of a loop that terminates when a fixed point is reached. The loop consists of the following steps:

1. redundant comparison substitution e.g., the comparison `cmp gt 1 0` is replaced with the constant `true`.
2. redundant condition-branching substitution e.g., `cbr true block1() block2()` is replace with the `br block1()`.
3. redundant binary operation substitution e.g., `add 1 0` is replaced with the constant `1`.
4. redundant unary operation substitution e.g., `not true` is replaced with the constant `false`.

2.2.11 Memory to SSA Registers

This pass acts on functions' local variables that are of a type that fits in FuelVM's 64-bit registers, and that are only read and written directly, without any pointer arithmetic (that is, only used in `load` and `store` instructions). The `mem2reg` pass "promotes" these variables to SSA registers: this concretely means that every time some value is stored to the variable, this value replaces every subsequent `load` of the variable, until the next `store`. The purpose of this optimisation is to let the register allocation algorithm, used in the following steps of the compilation, better decide when the variable should be held in memory and when in the machine registers.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Instruction Side Effects Not Thoroughly Considered Acknowledged

5.1 Instruction Side Effects Not Thoroughly Considered

Correctness Low Version 1 Acknowledged

CS-FVSO-001

The `dce` pass relies on the function `may_have_side_effects()` to detect whether an unused instruction can be pruned out. However, the function does not consider that instructions like arithmetic operations may set some reserved registers (like `$of` for overflow), that may later be used, even though the result of the instruction itself is not. This can result in incorrectly pruning out instructions.

Acknowledged:

Fuel replied:

> We do consider that arithmetic operations do not have side effects and that overflows will cause reverts. It is not the case for *u64* so still some work to do here. We consider all invalid arithmetic operations Undefined Behavior at this time.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- FuelVM Instruction Variant Not Hashed Code Corrected

Low-Severity Findings	1

- Modulo Zero Uncaught in Constant Folding Code Corrected

6.1 FuelVM Instruction Variant Not Hashed

Correctness Medium Version 1 Code Corrected

CS-FVSO-004

The `fn-dedup` optimisation pass does not accumulate the instruction variant into the current hash, when it encounters a FuelVM instruction. This means, for example, that if two functions are identical except for one instruction (being `state_load_quad_word` in one function, and `state_store_quad_word` in the other), they will be incorrectly de-duplicated by this optimisation, and one of the two will be eventually eliminated altogether by dead code elimination.

Code corrected:

A discriminant has been added before the hashing of each instruction and each particular FuelVM instruction.

6.2 Modulo Zero Uncaught in Constant Folding

Correctness Low Version 1 Code Corrected

CS-FVSO-003

The `const-folding` optimisation pass aims at simplifying some instructions whose operands are all constants, by replacing them with the constant result.
When a modulo instruction is encountered that can be "folded" in this way, the expression is directly evaluated, without checking for the RHS to be non-zero. This causes a panic in the compiler, if a Sway program contains code like the following:

```
let a = 3 % 0;
```

Note that in the current version of the compiler, the optimization will only be triggered when the `--release` flag is used.

Code corrected:

A check on the denominator has been added to prevent divisions by 0.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Constant Demotion Generates Duplicate Variables

Informational **Version 1**

CS-FVSO-002

The `const-demotion` pass generates a new local variable for each *usage* of each constant (of a demotable type). This creates several duplicates of the same variable, if the same constant is used several times.
However, only the last of these duplicates ends up replacing all the usages of the constant. This is due to the source code line

```
replace_map.insert(c_val, load_val);
```

which repeatedly overwrites the same entry `c_val` with always-different `load_val` Values.
Therefore, the extra variables are effectively unused, and will be pruned by a subsequent DCE pass.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Impact of ABI Encoding V1

Note Version 1

Compiling a simple smart contract with ABI encoding v1, seems to lead to a large overhead, in terms of bytecode size and execution gas cost, compared to v0, around 5x-8x.

8.2 Insufficient Domain Separation in the Hashing Algorithm

Note Version 1

The function-hashing algorithm used in the `fn-dedup` pass does not properly separate the different components that make up the function to be hashed (e.g. blocks, variables, instructions); instead, every component is represented as a sequence of integers and then sequentially accumulated into a "global" hash.
This approach has a high likelihood of producing unintended collisions, in case the "second half" of a component's serialization can be equivalently interpreted as the "first half" of the serialization of the following component, or if two different components have the same serialization.
This latter case applies, for example, to localized block ids and value ids, both being progressive and independent integer identifiers: a given integer, accumulated into the hash, could therefore be either a block id or a value id. Another id, that is implicitly progressive and independent of the previous two, and therefore suffers from the same potential collision risk, is Rust's discriminant of an enum value: when encountering a FuelVM-type instruction, its discriminant is added to the global hash, so as to distinguish between the possible variants.