# ideco: A Framework for Improving Non-C Decompilation

Sam Lerner
Independent
Indianapolis, USA
lerner98@gmail.com

## Abstract

We introduce the *ideco* framework for improving the decompilation of non-C programming languages. *ideco* provides users with the ability to create rules which rewrite parts of the decompilation.

We show that by using a small set of rules, the number of lines of decompiled code for binaries written in C++, Swift, Go, and Rust can be decreased by 5% to 10%. In addition, by using GPT-4o and GPT-4.1-mini as test subjects, we show that a reverse engineering task is easier to solve when its decompilation is processed by *ideco*.

## CCS Concepts

• **Security and privacy → Software reverse engineering**.

## Keywords

Decompilation, Software Understanding, Reverse Engineering

**ACM Reference Format:**
Sam Lerner. 2025. ideco: A Framework for Improving Non-C Decompilation. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3733822.3764668

```swift
func f(d: [Int:Int]) {
    print("Entries:")
    var sum = 0

    for (k, v) in d {
        print("  \(k): \(v)")
        sum += v
    }

    let avg = Float(sum) / Float(d.count)
    print("Average: \(avg)")
}
```

**(a) Original Swift Code**

```
func f(arg1: [Int:Int]) -> void {
    var var_48: Int
    print("Entries:")
    var_48 = 0x0
    for (k, v) in arg1 {
        print("  \(k): \(v)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

**(b) ideco Recovered Code**

**Figure 1: The original source code (A) vs. the decompilation processed by ideco (B). The initial decompilation, at 156 lines, is too large to fit here and can be found in Appendix A. By defining rewrite rules and new data types, the size of the decompilation can be drastically reduced and more closely match the original source.**

## 1 Introduction

The use of abstractions is common in programming as it allows programmers to ignore many of the implementation details of their code. While a boon for programmer productivity, these abstractions need to be implemented in terms of lower-level primitives and can therefore produce large amounts of code. Furthermore, since this code is machine-generated, it is often unintuitive to a human and can therefore be difficult to reverse engineer.

Take, for example, the Swift code in Figure 1a which prints the entries in a `Dictionary` followed by the average of its values. While this function is only 10 Source Lines of Code (SLOC), the function when decompiled by Binary Ninja is 152 lines and 134 lines in Ghidra. This growth factor in SLOC places a burden on the reverse engineer to both (a) figure out which abstractions are used, and (b) store this information out of band – usually either in their head or in notes or comments.

Starting from the Binary Ninja decompilation in Appendix A, we are able to transform the code into Figure 1b using *ideco* rewrite rules. At 14 SLOC, this code is not only an order of magnitude shorter than the original decompilation, but line-by-line it's much closer to the original source, utilizing Swift features such as iterators, tuples, and string interpolation.

This allows an analyst who creates these rules to store their hard-earned information about the abstractions used in the decompilation itself, as opposed to having to mentally reconstruct it every time when viewing the function. In addition, *ideco* rules are easy to share and are often applicable to a wide range of binaries. Therefore, given a large enough set of pre-existing rules, the code in Figure 1b might automatically be fully or partially recovered and the analyst not have to learn nearly as much about the target language internals in order to reverse engineer the program.


*SURE '25, Taipei, Taiwan*

ACM ISBN 979-8-4007-1910-3/25/10
https://doi.org/10.1145/3733822.3764668

## 2 Related Work

There has been recent work towards the goal of improving non-C decompilation in Binary Ninja. Specifically, Binary Ninja recently released a feature known as Language Representations [4]. This allows a plugin author to create an adapter which takes a High Level Intermediate Language (HLIL) tree and produces decompilation which doesn't necessarily follow C syntax. This is a great step forward; however, there are some shortcomings with this approach that *ideco* addresses.

First, the amount of code needed for an adapter is large. For example, the implementation of the Rust language representation is almost 3000 lines [3].

One reason for this is that each token is printed with a separate function call. For example, to print a variable declaration with the format "`{type} {variable};`", one needs a minimum of four function calls: one to print the variable type, one for the space, one for the variable, and one for the semicolon. To contrast, *ideco* uses a domain-specific language (DSL) for rendering trees to strings and the statement can be printed with a single function call using the DSL string "`${variable.type} $variable;`".

Another reason is that printing a special case of the HLIL tree requires a manual traversal. For example, if we are trying to print the statement "`a = a + 1`" as "`a += 1`", then one needs a minimum of three checks: one that the statement is an assignment, another that the right-hand-side is an addition, and another to check that the left-hand-side of the assignment and the left-hand-side of the addition are the same expression. *ideco* solves this with the use of another DSL for matching sub-trees of the HLIL. Therefore, the above statement could be matched with the DSL string "`$var = $var + $rhs`" and printed as "`$var += $rhs`".

Furthermore, language adapters are ill-suited for handling sequences of HLIL nodes. For example, the following code could be generated for copying a struct:

```
struct_type b;
b.field0 = a.field0;
b.field1 = a.field1;
```

The problem stems from the fact that a block contains a sequence of statements. Normally, a block would be printed by simply iterating over its statements and printing each one with a newline in between. However, to print these three statements as "`struct_type b = a;`", this loop would need to be modified to check if every triple of statements matches the above pattern. Therefore, as more and more special cases of sequences of statements are handled, the print loop would get more and more complicated.

To solve this, *ideco* allows for sub-sequences of statements like the three above to be matched as a new statement type. Therefore, the code to print a block or sequence of statements is unchanged from the simple case.

## 3 System Overview

*ideco* recovers high-level code constructs by creating a tree of "descriptor" instances and applying a series of rewrite rules. A descriptor is a schema for a node in the concrete syntax tree (CST), defining the names and types of its children.

Each descriptor is also a rewrite rule as it defines a pattern in the CST from which to construct itself (the left-hand-side) as well as how to render the resulting node to text (the right-hand-side). When the left-hand-side of that rewrite rule is matched, an instance of that descriptor is created, using the children matched from the pattern.

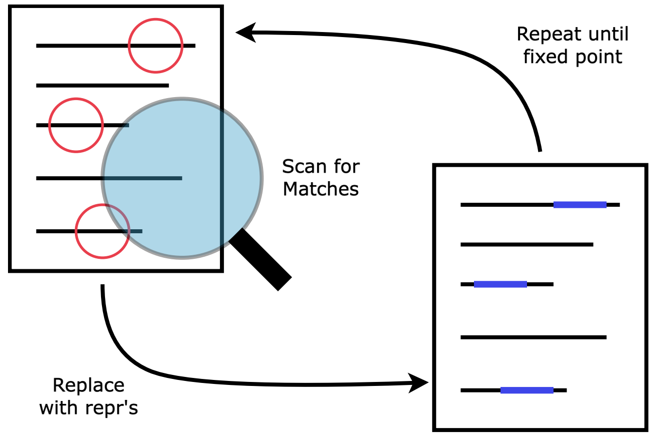


**Figure 2: The annotated decompilation is scanned for matching templates and the found candidates are substituted. This process is iterated until no new matches are found.**

*ideco* is implemented as a pass on top of Binary Ninja's HLIL and initially maps each node in the HLIL tree to a descriptor. These initial descriptors are an almost one-to-one mapping to the HLIL node types, e.g. the HLIL node `HLIL_IF` is mapped to the descriptor `hlil.If`. *ideco* then iterates over the descriptors which define a rewrite rule and attempts to match that rule in the current CST. If the rule is matched, it is then applied and the process is repeated until a fixed point is reached. An illustration of this process can be found in Figure 2.

## 4 System Implementation

A descriptor is simply a Python class which implements the following interface:

(1) A `match` function which specifies a subsection of the CST that the descriptor should be instantiated from.
(2) A `__repr__` function which renders the instantiated descriptor to a string.
(3) An optional `__init__` function which is called one time when a descriptor is instantiated.

There are other optional functions in this interface like `references` for generating x-refs; however, `match`, `__repr__`, and `__init__` are the most important for this paper.

The descriptors form an inheritance hierarchy where most HLIL nodes descend from either:

- `Stmt`: A statement in the HLIL tree such as an if, a block, or a loop.
- `Expr`: An expression such as a variable, a binary expression, or a function call.
- `DataType`: A data type represented in the decompilation.

Two example descriptors for HLIL nodes can be found in Figure 3a. The first descriptor is `Call` which represents a function call. The descriptor has two children, `func` and `args`, with types `Expr` and `list [Expr]` respectively. This means that when a `Call` is matched, `func` should be a sub-tree of the CST with a descriptor that inherits from `Expr` and `args` should be a sequence in the CST where each element descends from `Expr`.

By using the `Expr` type for the `func` child, we are allowing `Call` to match both direct calls (using a symbol name), and indirect calls (using a function pointer or variable). This is because in the direct case, `func` will be of type `Symbol` which is a subtype of `Expr` and in the indirect case, `func` will likely have type `Deref`, which descends from `Expr` via `UnaryExpr`. A full inheritance hierarchy for the HLIL descriptors can be found in Appendix B.

Descriptors in the `hlil` module don't need to specify a `match` function since they are created directly by *ideco*.

Descriptors are split into modules for both clarity and code re-use. For example, the `rc` module in Figure 3b imports the `hlil` module so that it can create descriptors which inherit from `hlil.Call`, `hlil.Stmt`, and `hlil.VarInit`. This inheritance is important as it tells the matching system that `ReferenceCountingOp`, for example, is a `Call`, and therefore also an `Expr`, and can be matched as the `rhs` of a `VarInit`.

Furthermore, `OpWithValue` inherits from `VarInit` and re-defines its `rhs` child as a `ReferenceCountingOp`. Therefore, the other annotations and `__repr__` function, don't have to be re-implemented. By specializing `VarInit` with `OpWithValue`, in its `__init__` method, we can propagate (replace all the uses of) the left-hand-side with the right-hand-side

Another aspect of the system is that when a descriptor's string representation evaluates to the empty string (as in `VoidOp`), that subtree in the CST is deemed to be "hidden" and will not be displayed in the rendered text (as in Figure 4).

```python
class Call(Expr):
    func: Expr
    args: list[Expr]

    def __repr__(self):
        return eval_repr('$func(${args*<, >})')

class VarInit(Stmt):
    lhs: Var
    rhs: Expr

    def __repr__(self):
        return eval_repr('${lhs.data_type} $lhs = $rhs')

```

**(a) Descriptors for Call and VarInit Statements in hlil module**

```python
import hlil

class ReferenceCountingOp(hlil.Call):
    @staticmethod
    def match():
        (ok, vars) = eval_tmpl('$func(${args*<, >})')

        return ok and vars['func'].name in {
            '_objc_retain', '_objc_release',
            '_swift_retain', '_swift_release',
            ...
        }

    def __repr__(self):
        return eval_repr('${args[0]}')

class VoidOp(hlil.Stmt):
    call: ReferenceCountingOp

    @staticmethod
    def match():
        return eval_tmpl('$call')[0]

    def __repr__(self):
        return eval_repr('')

class OpWithValue(hlil.VarInit):
    dt: hlil.DataType
    rhs: ReferenceCountingOp

    @staticmethod
    def match():
        return eval_tmpl('$dt $lhs = $rhs')[0]

    def __init__(self):
        self.lhs.propagate(self.rhs)

```

**(b) Descriptors in the Reference Counting (rc) module**

**Figure 3: Descriptors are formatted with a DSL combining string literals, keywords, child expressions, and sequences. Since descriptors can be inherited from, string representations and templates can be partially re-used and made more specific for certain situations.**

## 4.1 Matching in Detail

As described in Section 3, *ideco* attempts to apply descriptors which define a `match` function until no more matches are found. A `match` function will call `eval_tmpl` to evaluate if a template written in a domain-specific language (DSL) (described in Section 4.2) matches the decompilation at a given location. If successful, the variables bound by the template are returned so that can be used for further validation. This combined use of a DSL and regular Python code allows for simple rules to be written in such a way that it is obvious what they match since it's exactly the text of the decompilation (using the DSL) while providing rule writers with the ability to implement more complex logic when needed (using Python).

For example, in Figure 3b, `ReferenceCountingOp.match` first evaluates a template which corresponds to a generic function call. This descriptor should only match functions which perform reference counting operations and therefore subsequently checks if the name of the `func` bound variable is in a known set of function names.

If the `match` function returns true, the bound variables (e.g. `func` and `args`) are checked to descend from the descriptors in the annotations (`Expr` and `list[Expr]`), and if they do, the descriptor is instantiated. For sequences such as `args`, each element is checked to descend from the inner type (`Expr`).

## 4.2 Template Language

The use of a domain-specific language (DSL) to specify templates is for ergonomics. If this DSL did not exist, the rule writer would

```
int64_t a = 1;
_objc_release(b);
int64_t c = 2;
```

**(a) Original**

```
int64_t a = 1;

int64_t c = 2;
```

**(b) After matching**

```
int64_t a = 1;
int64_t c = 2;
```

**(c) After filtering**

**Figure 4: The call to _objc_release (A) matches the VoidOp descriptor (B). Since this descriptor produces no text, it is then removed from the parent sequence of statements (C).**

have to walk the CST and manually check the descriptor for each node and its children.

Not only would this be a cumbersome amount of code, but it would be necessary to expose the CST, instead of the text of the decompilation, to the rule writer.

Take, for example, the decompilation `void* foo = bar`. According to the rules previously shown, the underlying descriptor for this string could be a `hlil.VarInit` or an `rc.OpWithValue` since they share a string representation. The only way to know the difference is to look at the underlying CST.

With templates, if we'd like to write a rule which displays variable declarations with Swift syntax, we could create a new descriptor whose `match` function evaluates the template `"$dt $lhs = $rhs"` where `$rhs` has the type `hlil.Expr` and whose `__repr__` function uses the format ``"`var` $lhs: $dt = $rhs"``. This template does not need to know whether or not the matched substring is an `hlil.VarInit` or an `rc.OpWithValue` and both forms can be matched with the same rule.

The full grammar for the template language is defined in Table 1.

**Table 1: Template Language Specification**

| | | |
|---|---|---|
| <template> | ::= | <piece>* |
| <piece> | ::= | <literal> \| <var> \| <seq> |
| <var> | ::= | **$**<ident> |
| <seq> | ::= | **${**ident*<separator>**}** |
| <separator> | ::= | <literal> |
| <ident> | ::= | [A-Za-z0-9_]+ |
| <literal> | ::= | [ˆ$]+ |

## 4.3 Repr Language

A descriptor's `__repr__` method also uses a DSL. This is again provided for programmer ergonomics. The grammar of the repr language is almost identical to that of the template language with the addition of keywords and attributes as defined in Table 2.

**Table 2: Repr Language Specification**

| | | |
|---|---|---|
| <piece> | ::= | ... \| <keyword> \| <attr> |
| <keyword> | ::= | **'**<literal>**'** |
| <attr> | ::= | **${**path**}** |
| <path> | ::= | <comp>(**.**<comp>)+ |
| <comp> | ::= | <child> \| <child>**[**<index>**]** |
| <child> | ::= | <ident> |
| <index> | ::= | [1-9][0-9]* |

```
int64_t rax_13
int64_t rdx_6
rax_13, rdx_6 = DefaultStringInterpolation.init(0, 2)
s.q = rax_13
s:8.q = rdx_6
```

**(a) Original Decompilation**

```
ValueType rax_13 = DefaultStringInterpolation.init(0, 2)
s = rax_13
```

**(b) ideco-Processed Decompilation**

**Figure 5: Existing decompilers cannot handle when a variable is distributed across multiple registers (A). Using ideco rules, we can convert this code into something which more closely resembled the original source (B).**

The reason that the repr DSL provides facilities for accessing sub-attributes and sequence indexes, as in `"${lhs.data_type}"` is because components of the final string need to be evaluated in the order in which they appear.

This is because the matching system needs to know the bounds of every node in the final text as well as the descriptor at each location. If, for example, in `VarInit.__repr__`, we executed the code:

```
eval_repr(f'{self.lhs.data_type} {self.lhs} = {self.rhs}')
```

all the expressions in the f-string would be first converted into strings and the parsed DSL program would be one literal. The matching system would therefore not be able to recognize where the `hlil.DataType`'s and `hlil.Expr`'s are in order to match `rc.OpWithValue`.

## 4.4 Dynamic Rules

Since the descriptors which define the rewrite rules are Python classes, descriptors can be dynamically generated. This is particularly useful when dealing with certain optimizations for value types produced by some compilers.

While languages like C and C++ primarily pass objects around as pointers and references for efficiency, modern languages such as Rust or Swift make more heavy use of passing objects by value. This passing style; however, can negatively impact performance since it requires objects larger than the size of a register to be copied every time they are moved. Therefore, some modern compilers implement an optimization for the x86-64 architecture which places objects of 16 bytes or fewer into the register pair (RAX, RDX). This optimization means that moving such an object only requires copying two registers which is much more efficient than calling `memcpy` or `memmove`.

Today's decompilers, however, are ill-equipped to handle such code and produce decompilation like Figure 5a. Furthermore, even when an analyst realizes that the (RAX, RDX) pair corresponds to a single variable, they are unable to modify the decompilation to reflect this as current decompilers only allow for variables to be located in a single register or in some contiguous region of memory (likely on the stack).

In *ideco*, variables are no different from other descriptors and can therefore be represented by any subsection of the CST. Figure 6 shows how this can be accomplished to produce the code in Figure 5b.

```
class ValueTypeInit(hlil.VarInit):
    dt1: hlil.DataType
    dt2: hlil.DataType
    var2: hlil.Var
    rhs: hlil.Call

    @staticmethod
    def match():
        return eval_tmpl('''
            $dt1 $lhs
            $dt2 $var2
            $lhs, $var2 = $rhs
        ''')[0]

    def __init__(self):
        dt1 = self.lhs.data_type
        dt2 = self.var2.data_type

        dt = create_type('ValueType')
        self.lhs.set_type(dt)

        def match_copy():
            (ok, vars) = eval_tmpl('''
                 $dst1 = $field1
                 $dst2 = $field2
            ''')
            return ok and \
                 vars['field1'].data_type == dt and \
                 vars['field2'].data_type == dt2

        def init_copy(self):
             self.dst1.set_type(dt)
             self.dst2.set_type(dt2)

        create_descriptor(
            name=f'{dt}_Copy',
            match=match_copy,
            init=init_copy,
            repr=lambda: eval_repr('$dst1 = $field1'),
            annos={'dst1':hlil.Expr, 'field1':dt1, ...},
        )
```

**Figure 6: Multiple statements can be matched by a descriptor. In their init methods, descriptors can also create new descriptors from the bound variables.**

As with `OpWithValue`, the `ValueTypeInit` descriptor inherits from the `hlil.VarInit` and specializes the `rhs` child.

In `ValueTypeInit.match`, we see how a sequence can be matched simply by writing a template with the identical syntax of the CST. In this case, since we are matching statements, we separate them by newlines, but if we were matching call parameters, we would separate them by commas. The matching system will detect that the matched text corresponds to multiple statements and replace that sub-sequence of the parent sequence with the new descriptor.

In `ValueTypeInit.__init__`, the data types of the two struct fields (`lhs` and `var2`) are used to dynamically create the `ValueType_Copy` descriptor. This descriptor matches a pair of assign statements and checks if the types of the right-hand-sides correspond to the types or the new struct. Then in `ValueType_Copy.__init__` method, the types of the right-hand-sides are propagated to the left-hand-sides so that any subsequent copies of this new variable will also be recognized.

| Language | SLOC Before | SLOC After | Avg % Decrease |
|---|---|---|---|
| C++ | 72.665 | 71.437 | 5.231 |
| Swift | 61.713 | 56.798 | 9.458 |
| Rust | 73.667 | 65.333 | 8.819 |
| Go | 55.933 | 52.067 | 7.136 |

**Table 3: The change in SLOC of the decompilation when applying a limited set of rewrite rules. Even though the rules were not created with Go and Rust in mind, they are partially applicable to those languages as well.**

## 5 Evaluation

Evaluating decompilation is a difficult problem since the intended audience is most often human beings with varying experiences, preferences, and aptitudes. In addition, metrics which have been generally seen as good proxies for good decompilation, such as the number of goto's, have been disputed [1] and can be desirable in some circumstances while undesirable in others.

Therefore, we use two different methods to evaluate *ideco*: the number of lines in the resulting decompilation and how well different large language models (LLM's) performs on a reverse engineering task.

### 5.1 Code Size

The first metric we use is simply the average number of lines of each decompiled function. While certainly crude, we believe this is a good rough metric since much of the difficulty in reverse engineering higher-level languages comes from the sheer amount of code that they produce. In addition, since the number of bugs in code is proportional to the number of lines [2], it is reasonable to suspect that the number of errors that a reverse engineer makes is proportional to the number of lines of decompiled code.

The results in Table 3 show the change in average lines of code per-function when applying a small set of rules to binaries written in different programming languages. The rules applied were the reference-counting and value type rules previously shown, as well as a rule to remove destructor calls.

The Swift and C++ binaries were each chosen as random samples of 100 binaries on a 2015 MacBook Pro which link libc++ and any of the Swift standard libraries, respectively. All binaries were compiled for the x86-64 architecture.

We only evaluated three Go and three Rust binaries in order to test the generalizability of the rules. These binaries were small crackme's generated by GPT-4o. Even though the rules were written either in a language-agnostic manner or for Swift/C++, they provide a similar improvement in the decompilation size for Go and Rust. This is primarily due to the rule for value types since the Go and Rust compilers perform similar optimizations as the Swift compiler.

While a change in average function length of a couple lines might not seem earth-shattering, we show in the next section how these rules help different LLM's solve a Swift crackme. In addition, it's intended for more language-specific rules to be written which should not only further decrease the code size, but recover more source-language constructs to aid in program understanding. This can be seen for example in Figure 1b where bespoke rules were written to achieve and smaller and more high-level decompilation.

## 5.2 LLM Evaluation

As a more qualitative evaluation, we chose to evaluate GPT-4o and GPT-4.1-mini's performance on solving a simple Swift crackme. This is meant to serve as both a proxy for human performance on such a test, as well as seeing how *ideco* can improve the automated analysis of decompilation.

The crackme first calculates the SHA256 hash of the hardcoded string "foobar". It then reads the attempted solution from stdin, manually reverses that string (without using the `reversed` method), and compares its SHA256 hash against the value computed for "foobar". If the hashes are equal, a message stating that the crackme has been solved is printed. The full code can be found in Appendix C.

Each experiment began with the following prompt:

> Here is the decompilation of a crackme. Write pseudocode for each function and provide a solution to the crackme. <decompilation inserted here>

Every time the AI showed its reasoning to be incorrect, it was informed as such. However, we never told the AI *what* was wrong, only *where* its understanding was flawed. An example is as follows:

> GPT:
> *The validate function takes the user input, hashes it unchanged, and compares it with the expected hash. <incorrect pseudocode>*
>
> Prompter:
> Your understanding of the validate function is incorrect. Please re-read the decompilation and try again.

Each experiment was run 20 times with the same prompting technique and was limited to 10 "steps". We define a step as one (prompt, response) pair. If the model did not find a solution in 10 steps, the run was terminated and deemed a failure. The results are as follows:

| GPT-4o | | | |
|---|---|---|---|
| Version | Interp Rate | Solve Rate | Avg # of Steps |
| Baseline | 85% | 75% | 4.13 |
| ValueTypes | 90% | 90% | 3.28 |
| ValueTypes + Swift | 100% | 100% | 1.85 |

| GPT-4.1-mini | | | |
|---|---|---|---|
| Version | Interp Rate | Solve Rate | Avg # of Steps |
| Baseline | 5% | 5% | 5.0 |
| ValueTypes | 95% | 95% | 3.47 |
| ValueTypes + Swift | 100% | 100% | 1.05 |

**Table 4: Results of the LLM user study. Providing clearer and more concise code increases the accuracy and speed of finding the solution.**

The baseline version of the code is the HLIL output from Binary Ninja whereas the `ValueTypes` version includes the rules for cleaning up the register-in-structs optimization found in Figure 6. The final version also includes the `Swift` module which contains rules for Swift-specific functionality such as iterators and string interpolation. The full decompilation for each version can be found in Appendix D

While the solve rate is the rate with which the model was able to come up with the correct solution to the crackme ("raboof"), the "Interp Rate" is the rate with which the model figured out that the user input is reversed before it's hashed.

With only the `ValueTypes` rules, the success rate is increased by 15% for GPT-4o. We hypothesize two reasons for this: 1) these rules do the most to decrease code size, which is important so that more of the code can fit into the LLM's context window, and 2) these rules are able to define and use data types which makes the data flow of the program easier to follow.

It seems likely that the amount of code, and hence the amount of the context window taken up, is a crucial factor since GPT-o3, which has a much larger window, does not seem to be affected by the different versions of the code and reliably solves the crackme every time.

Further studies with human subjects should to be conducted as the criteria for improving human performance is unlikely to be exactly the same as improving LLM performance. This is, however, an interesting signal that decompilation processed by *ideco* might be more intelligible by a human analyst as well.

# 6 Limitations

There are two main drawbacks to the approach presented in this paper:

(1) Brittleness: The approach in *ideco* is more or less to reconstruct each high-level abstraction individually. This means that when a new abstraction is encountered, new rules will be needed to reverse it. For example, in the decompiled code in Appendix A, 28 new rules were created to recover the Swift constructs that the program uses. While we would expect to be able to re-use many of these rules for other Swift code, it was still an unwieldy, albeit relatively straightforward process.
This brittleness, or non-generalizability, of rules also extends to variations in compiled code due to instruction scheduling, register allocation, and other compiler optimizations.

(2) Performance: The template language is first and foremost based on matching text. This is for programmer ergonomics and readability of DSL code. However, this comes with a performance cost since the full decompilation needs to be scanned for each rule. In addition, features of the template language that provide for rule flexibility such as matching sequences, are inherently bad for performance as they require heavy use of backtracking to implement.
Since each rule has relatively bad performance and many rules are needed, *ideco* is slow and is not currently feasible to be used in the critical path of a decompiler. That being said, there are many opportunities for performance improvements

and the authors believe that *ideco* can be made to have usable performance.

## 7 Conclusion

We present *ideco*, a framework for creating and applying rewrite rules to decompilation. We show how these rules can reconstruct higher-level constructs from C-style decompilation and remove much of the bloat associated with the decompilation of non-C languages.

Furthermore, we show, using code size as a proxy, and with a novel LLM user study, that these rules make decompilation more intelligible and easier to analyze.

## References


[1] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation. In *Proceedings of the USENIX Security Symposium*.

[2] Steve McConnell. 2004. Code Complete. In *Code Complete: A Practical Handbook of Software Construction*. https://archive.org/details/code-complete-2nd-edition

[3] et al. Peter LaFosse. 2024. Pseudorust Implementation. In *Binary Ninja Blog*. https://github.com/Vector35/binaryninja-api/blob/dev/lang/rust/pseudorust.cpp

[4] Jordan Wiens. 2024. 4.2 Frogstar. In *Binary Ninja Blog*. https://binary.ninja/2024/11/20/4.2-frogstar.html


## A Example Code Decompilation

The original Binary Ninja decompilation for the function in Figure 1a is as follows:

```
int64_t f(d:)(void* arg1) {
    void* var_40
    &_memset(&var_40, 0x0, 0x8)
    int64_t var_48
    &_memset(&var_48, 0x0, 0x8)
    void var_70
    &_memset(&var_70, 0x0, 0x28)
    float var_90
    &_memset(&var_90, 0x0, 0x4)
    int64_t var_a0
    &_memset(&var_a0, 0x0, 0x10)
    int64_t k_2
    &_memset(&k_2, 0x0, 0x8)
    int64_t v_2
    &_memset(&v_2, 0x0, 0x8)
    int64_t var_c8
    &_memset(&var_c8, 0x0, 0x10)
    var_40 = arg1
    int64_t rax
    int64_t* rdx
    rax, rdx = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    int64_t rax_1
    int64_t rdx_1
    rax_1, rdx_1 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
    rdx[0x3] = type metadata for String
    *(rdx) = rax_1
    rdx[0x1] = rdx_1
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    int64_t rax_4
    int64_t rdx_3
    rax_4, rdx_3 = &default argument 1 of print(_:separator:terminator:)()
    int64_t rax_5
    int64_t rdx_4
    rax_5, rdx_4 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rdx_3, rax_5, rdx_4)
    &_swift_bridgeObjectRelease(rdx_4)
    &_swift_bridgeObjectRelease(rdx_3)
    &_swift_bridgeObjectRelease(rax_3)
    var_48 = 0x0
    &_swift_bridgeObjectRetain(arg1)
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            &outlined destroy of [Int : Int].Iterator(&var_70)
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            var_90 = zmm0_1
            int64_t rax_29
            int64_t* rdx_20
            rax_29, rdx_20 = &_allocateUninitializedArray<A>(_:)(0x1)
            int64_t rax_30
            int64_t rdx_21
            rax_30, rdx_21 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            var_a0 = rax_30
            int32_t var_204 = 0x1
            void* rax_31
            void* rdx_22
            rax_31, rdx_22 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Average: ", 0x9, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&var_a0, rax_31, rdx_22)
            &_swift_bridgeObjectRelease(rdx_22)
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_a0, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            void* rax_32
            void* rdx_24
            rax_32, rdx_24 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&var_a0, rax_32, rdx_24)
            &_swift_bridgeObjectRelease(rdx_24)
            int64_t rax_33 = var_a0
            &_swift_bridgeObjectRetain(rdx_21)
            &outlined destroy of DefaultStringInterpolation(&var_a0)
            int64_t rax_34
            int64_t rdx_25
```

```
            rax_34, rdx_25 = &String.init(stringInterpolation:)(rax_33, rdx_21)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            rdx_20[0x1] = rdx_25
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            int64_t rax_37
            int64_t rdx_27
            rax_37, rdx_27 = &default argument 1 of print(_:separator:terminator:)()
            int64_t rax_38
            int64_t rdx_28
            rax_38, rdx_28 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rdx_27, rax_38, rdx_28)
            &_swift_bridgeObjectRelease(rdx_28)
            &_swift_bridgeObjectRelease(rdx_27)
            return &_swift_bridgeObjectRelease(rax_36)
        }
        k_2 = k
        v_2 = v
        int64_t rax_12
        int64_t* rdx_7
        rax_12, rdx_7 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        int64_t rax_13
        int64_t rdx_8
        rax_13, rdx_8 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        var_c8 = rax_13
        void* rax_14
        void* rdx_9
        rax_14, rdx_9 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&var_c8, rax_14, rdx_9)
        &_swift_bridgeObjectRelease(rdx_9)
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_c8, &k_1, type metadata for Int, protocol witness table for Int)
        void* rax_16
        void* rdx_11
        rax_16, rdx_11 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(": ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&var_c8, rax_16, rdx_11)
        &_swift_bridgeObjectRelease(rdx_11)
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_c8, &v_1, type metadata for Int, protocol witness table for Int)
        void* rax_18
        void* rdx_13
        rax_18, rdx_13 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&var_c8, rax_18, rdx_13)
        &_swift_bridgeObjectRelease(rdx_13)
        int64_t rax_19 = var_c8
        &_swift_bridgeObjectRetain(rdx_8)
        &outlined destroy of DefaultStringInterpolation(&var_c8)
        int64_t rax_20
        int64_t rdx_14
        rax_20, rdx_14 = &String.init(stringInterpolation:)(rax_19, rdx_8)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        rdx_7[0x1] = rdx_14
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        int64_t rax_23
        int64_t rdx_16
        rax_23, rdx_16 = &default argument 1 of print(_:separator:terminator:)()
        int64_t rax_24
        int64_t rdx_17
        rax_24, rdx_17 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rdx_16, rax_24, rdx_17)
        &_swift_bridgeObjectRelease(rdx_17)
        &_swift_bridgeObjectRelease(rdx_16)
        &_swift_bridgeObjectRelease(rax_22)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

Each call to `print` is expanded into many calls which construct the Swift strings and perform the string interpolation. Below is a description of each rule created to produce the code in Figure 1b as well as the decompilation after applying said rule.

*A.0.1 LargeValueTypeInit.* Detects value types constructed on the stack and then initialized to zero with `memset`. This is a dynamic rule which then sets the type of the declared variable and creates the following dynamic rules:

- `LargeValueTypeCopy`: Detects a call to `memcpy` with the source being a value type, converts it into a regular assign statement, and propagates the type to the destination. Dead-code elimination is also performed on the variables of this type class.

```
int64_t f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    Type_5 var_a0
    Type_8 var_c8
    int64_t rax
    int64_t* rdx
    rax, rdx = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    int64_t rax_1
    int64_t rdx_1
    rax_1, rdx_1 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
    rdx[0x3] = type metadata for String
    *(rdx) = rax_1
    rdx[0x1] = rdx_1
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    int64_t rax_4
    int64_t rdx_3
```

```
    rax_4, rdx_3 = &default argument 1 of print(_:separator:terminator:)()
    int64_t rax_5
    int64_t rdx_4
    rax_5, rdx_4 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rdx_3, rax_5, rdx_4)
    &_swift_bridgeObjectRelease(rdx_4)
    &_swift_bridgeObjectRelease(rdx_3)
    &_swift_bridgeObjectRelease(rax_3)
    var_48 = 0x0
    &_swift_bridgeObjectRetain(arg1)
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            &outlined destroy of [Int : Int].Iterator(&var_70)
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            int64_t rax_29
            int64_t* rdx_20
            rax_29, rdx_20 = &_allocateUninitializedArray<A>(_:)(0x1)
            int64_t rax_30
            int64_t rdx_21
            rax_30, rdx_21 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            var_a0 = rax_30
            int32_t var_204 = 0x1
            void* rax_31
            void* rdx_22
            rax_31, rdx_22 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Average: ", 0x9, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&var_a0, rax_31, rdx_22)
            &_swift_bridgeObjectRelease(rdx_22)
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_a0, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            void* rax_32
            void* rdx_24
            rax_32, rdx_24 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&var_a0, rax_32, rdx_24)
            &_swift_bridgeObjectRelease(rdx_24)
            int64_t rax_33 = var_a0
            &_swift_bridgeObjectRetain(rdx_21)
            &outlined destroy of DefaultStringInterpolation(&var_a0)
            int64_t rax_34
            int64_t rdx_25
            rax_34, rdx_25 = &String.init(stringInterpolation:)(rax_33, rdx_21)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            rdx_20[0x1] = rdx_25
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            int64_t rax_37
            int64_t rdx_27
            rax_37, rdx_27 = &default argument 1 of print(_:separator:terminator:)()
            int64_t rax_38
            int64_t rdx_28
            rax_38, rdx_28 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rdx_27, rax_38, rdx_28)
            &_swift_bridgeObjectRelease(rdx_28)
            &_swift_bridgeObjectRelease(rdx_27)
            return &_swift_bridgeObjectRelease(rax_36)
        }
        int64_t rax_12
        int64_t* rdx_7
        rax_12, rdx_7 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        int64_t rax_13
        int64_t rdx_8
        rax_13, rdx_8 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        var_c8 = rax_13
        void* rax_14
        void* rdx_9
        rax_14, rdx_9 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&var_c8, rax_14, rdx_9)
        &_swift_bridgeObjectRelease(rdx_9)
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_c8, &k_1, type metadata for Int, protocol witness table for Int)
        void* rax_16
        void* rdx_11
        rax_16, rdx_11 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(": ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&var_c8, rax_16, rdx_11)
        &_swift_bridgeObjectRelease(rdx_11)
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_c8, &v_1, type metadata for Int, protocol witness table for Int)
        void* rax_18
        void* rdx_13
        rax_18, rdx_13 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&var_c8, rax_18, rdx_13)
        &_swift_bridgeObjectRelease(rdx_13)
        int64_t rax_19 = var_c8
        &_swift_bridgeObjectRetain(rdx_8)
```

```
        &outlined destroy of DefaultStringInterpolation(&var_c8)
        int64_t rax_20
        int64_t rdx_14
        rax_20, rdx_14 = &String.init(stringInterpolation:)(rax_19, rdx_8)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        rdx_7[0x1] = rdx_14
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        int64_t rax_23
        int64_t rdx_16
        rax_23, rdx_16 = &default argument 1 of print(_:separator:terminator:)()
        int64_t rax_24
        int64_t rdx_17
        rax_24, rdx_17 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rdx_16, rax_24, rdx_17)
        &_swift_bridgeObjectRelease(rdx_17)
        &_swift_bridgeObjectRelease(rdx_16)
        &_swift_bridgeObjectRelease(rax_22)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.2 SmallValueTypeInit.* Described in Figure 6, detects small value types in the (RAX, RDX) register pair. Creates the following dynamic rules:

- `SmallValueTypeCopy`: Detects when the register pair is copied to another pair of registers or stack locations. This dual assign is then propagated to the sites of its uses.
- `SmallValueTypeUse`: Detects the register pair in a sequence such as call parameters.
- `SmallValueTypePartialCopy`: Detects assigns to just a single register of the pair and propagates the destination.

```
int64_t f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    Type_14 rax_30
    Type_21 rax_13
    Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    Type_10 rax_1 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
    rdx[0x3] = type metadata for String
    *(rdx) = rax_1
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_11 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_12 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    &_swift_bridgeObjectRelease(rdx_4)
    &_swift_bridgeObjectRelease(rdx_3)
    &_swift_bridgeObjectRelease(rax_3)
    var_48 = 0x0
    &_swift_bridgeObjectRetain(arg1)
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            &outlined destroy of [Int : Int].Iterator(&var_70)
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_13 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_14 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            Type_15 rax_31 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Average: ", 0x9, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_31)
            &_swift_bridgeObjectRelease(rdx_22)
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_30, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            Type_16 rax_32 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_32)
            &_swift_bridgeObjectRelease(rdx_24)
            &_swift_bridgeObjectRetain(rdx_21)
            &outlined destroy of DefaultStringInterpolation(&rax_30)
            Type_17 rax_34 = &String.init(stringInterpolation:)(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_18 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_19 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            &_swift_bridgeObjectRelease(rdx_28)
            &_swift_bridgeObjectRelease(rdx_27)
            return &_swift_bridgeObjectRelease(rax_36)
        }
        Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        Type_21 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
```

```
        Type_22 rax_14 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_14)
        &_swift_bridgeObjectRelease(rdx_9)
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &k_1, type metadata for Int, protocol witness table for Int)
        Type_23 rax_16 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(": ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_16)
        &_swift_bridgeObjectRelease(rdx_11)
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &v_1, type metadata for Int, protocol witness table for Int)
        Type_24 rax_18 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_18)
        &_swift_bridgeObjectRelease(rdx_13)
        &_swift_bridgeObjectRetain(rdx_8)
        &outlined destroy of DefaultStringInterpolation(&rax_13)
        Type_25 rax_20 = &String.init(stringInterpolation:)(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_26 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_27 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        &_swift_bridgeObjectRelease(rdx_17)
        &_swift_bridgeObjectRelease(rdx_16)
        &_swift_bridgeObjectRelease(rax_22)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.3 ReferenceCountingOp.* Described in Figure 3b, removes reference counting operations such as `_swift_bridgeObjectRetain` and `_swift_bridgeObjectRelease`.

```
void f(d:)(void* arg1) {
    Type_21 var_48
    Type_22 var_70
    Type_6 rax_30
    Type_13 rax_13
    Type_1 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    Type_2 rax_1 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
    rdx[0x3] = type metadata for String
    *(rdx) = rax_1
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_3 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_4 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            &outlined destroy of [Int : Int].Iterator(&var_70)
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_5 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_6 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            Type_7 rax_31 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Average: ", 0x9, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_31)
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_30, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            Type_8 rax_32 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_32)
            &outlined destroy of DefaultStringInterpolation(&rax_30)
            Type_9 rax_34 = &String.init(stringInterpolation:)(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_10 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_11 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        Type_12 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
```

```
        Type_13 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        Type_14 rax_14 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_14)
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &k_1, type metadata for Int, protocol witness table for Int)
        Type_15 rax_16 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(": ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_16)
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &v_1, type metadata for Int, protocol witness table for Int)
        Type_16 rax_18 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_18)
        &outlined destroy of DefaultStringInterpolation(&rax_13)
        Type_17 rax_20 = &String.init(stringInterpolation:)(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_18 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_19 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

#### A.0.4 *RAIIOp.* 

Similar to `ReferenceCountingOp`, removes calls to object destructors.

```
void f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    Type_14 rax_30
    Type_21 rax_13
    Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    Type_10 rax_1 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
    rdx[0x3] = type metadata for String
    *(rdx) = rax_1
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_11 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_12 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_13 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_14 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            Type_15 rax_31 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Average: ", 0x9, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_31)
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_30, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            Type_16 rax_32 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_32)
            Type_17 rax_34 = &String.init(stringInterpolation:)(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_18 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_19 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        Type_21 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        Type_22 rax_14 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_14)
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &k_1, type metadata for Int, protocol witness table for Int)
```

```
        Type_23 rax_16 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(": ", 0x2, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_16)
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &v_1, type metadata for Int, protocol witness table for Int)
        Type_24 rax_18 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100003f7f, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_18)
        Type_25 rax_20 = &String.init(stringInterpolation:)(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_26 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_27 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

#### A.0.5 *StringInit.*

Detects calls to `String.init(_builtinStringLiteral:...)` and replaces it with the passed literal then sets the type of the left-hand-side to `String` (a value type).

```
void f(d:)(void* arg1) {
    Type_21 var_48
    Type_22 var_70
    Type_6 rax_30
    Type_13 rax_13
    Type_1 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    String rax_1 = "Entries:"
    rdx[0x3] = type metadata for String
    *(rdx) = rax_1
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_3 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_4 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_5 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_6 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            String rax_31 = "Average: "
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_31)
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_30, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            String rax_32 = ""
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, rax_32)
            Type_9 rax_34 = &String.init(stringInterpolation:)(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_10 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_11 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        Type_12 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        Type_13 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        String rax_14 = "  "
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_14)
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &k_1, type metadata for Int, protocol witness table for Int)
        String rax_16 = ": "
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_16)
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &v_1, type metadata for Int, protocol witness table for Int)
        String rax_18 = ""
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_18)
        Type_17 rax_20 = &String.init(stringInterpolation:)(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_18 rax_23 = &default argument 1 of print(_:separator:terminator:)()
```

```
        Type_19 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

### A.0.6 *StringAssignProp.* Propagates assignments of type `String`.

```
void f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    Type_14 rax_30
    Type_21 rax_13
    Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    rdx[0x3] = type metadata for String
    *(rdx) = "Entries:"
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_11 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_12 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_13 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_14 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, "Average: ")
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_30, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, "")
            Type_17 rax_34 = &String.init(stringInterpolation:)(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_18 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_19 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        Type_21 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, "  ")
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &k_1, type metadata for Int, protocol witness table for Int)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, ": ")
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &v_1, type metadata for Int, protocol witness table for Int)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, "")
        Type_25 rax_20 = &String.init(stringInterpolation:)(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_26 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_27 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

### A.0.7 *StringInterpInit.* Detects calls to the string constructor `String.init(stringInterpolation:)` and transforms it into a regular constructor call.

```
void f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    Type_14 rax_30
    Type_21 rax_13
    Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    rdx[0x3] = type metadata for String
    *(rdx) = "Entries:"
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_11 rax_4 = &default argument 1 of print(_:separator:terminator:)()
```

```
    Type_12 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_13 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_14 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, "Average: ")
            float var_a8 = zmm0_1
            &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_30, &var_a8, type metadata for Float, protocol witness table for Float, protocol witness table for Float)
            &DefaultStringInterpolation.appendLiteral(_:)(&rax_30, "")
            String rax_34 = String(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_18 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_19 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        Type_21 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, "  ")
        int64_t k_1 = k
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &k_1, type metadata for Int, protocol witness table for Int)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, ": ")
        int64_t v_1 = v
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &v_1, type metadata for Int, protocol witness table for Int)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, "")
        String rax_20 = String(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_26 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_27 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.8 InterpOp.* Groups calls to `DefaultStringInterpolation.appendLiteral` and `DefaultStringInterpolation.appendInterpolation` into one descriptor.

```
void f(d:)(void* arg1) {
    Type_21 var_48
    Type_22 var_70
    Type_6 rax_30
    Type_13 rax_13
    Type_1 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    rdx[0x3] = type metadata for String
    *(rdx) = "Entries:"
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_3 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_4 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            Type_5 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            Type_6 rax_30 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x9, 0x1)
            int32_t var_204 = 0x1
            rax_30.append("Average: ")
            float var_a8 = zmm0_1
```

```
            rax_30.append(var_a8)
            rax_30.append("")
            String rax_34 = String(rax_30)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_10 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_11 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        Type_12 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        Type_13 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x4, 0x2)
        rax_13.append("  ")
        int64_t k_1 = k
        rax_13.append(k_1)
        rax_13.append(": ")
        int64_t v_1 = v
        rax_13.append(v_1)
        rax_13.append("")
        String rax_20 = String(rax_13)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_18 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_19 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.9 StringInterpConstruct.* Detects a sequence of `InterpOp`'s delimited by calls to the `DefaultStringInterpolation` constructor and a string constructor (e.g. `StringInterpInit`). The operands from the `InterpOp`'s are then concatenated into a string in the output. The type of the output is set to a string and the interpolation propagated to the usage sites. Non-`InterpOp`'s in between the delimiting calls are hoisted above the output.

```
void f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    Type_14 rax_30
    Type_21 rax_13
    Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    rdx[0x3] = type metadata for String
    *(rdx) = "Entries:"
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_11 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_12 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            float var_a8 = zmm0_1
            int32_t var_204 = 0x1
            Type_13 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            String rax_34 = "Average: \(var_a8)"
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = rax_34
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_18 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_19 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        int64_t v_1 = v
        int64_t k_1 = k
        Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        String rax_20 = "  \(k_1): \(v_1)"
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = rax_20
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_26 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_27 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.10 InterpConstructProp.* Propagates interpolated strings.

```
void f(d:)(void* arg1) {
```

```
    Type_21 var_48
    Type_22 var_70
    Type_1 rax = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
    rdx[0x3] = type metadata for String
    *(rdx) = "Entries:"
    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(rax)
    Type_3 rax_4 = &default argument 1 of print(_:separator:terminator:)()
    Type_4 rax_5 = &default argument 2 of print(_:separator:terminator:)()
    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            float var_a8 = zmm0_1
            int32_t var_204 = 0x1
            Type_5 rax_29 = &_allocateUninitializedArray<A>(_:)(0x1)
            rdx_20[0x3] = type metadata for String
            *(rdx_20) = "Average: \(var_a8)"
            int64_t rax_36 = &_finalizeUninitializedArray<A>(_:)(rax_29)
            Type_10 rax_37 = &default argument 1 of print(_:separator:terminator:)()
            Type_11 rax_38 = &default argument 2 of print(_:separator:terminator:)()
            &print(_:separator:terminator:)(rax_36, rax_37, rax_38)
            return
        }
        int64_t v_1 = v
        int64_t k_1 = k
        Type_12 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
        rdx_7[0x3] = type metadata for String
        *(rdx_7) = "  \(k_1): \(v_1)"
        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
        Type_18 rax_23 = &default argument 1 of print(_:separator:terminator:)()
        Type_19 rax_24 = &default argument 2 of print(_:separator:terminator:)()
        &print(_:separator:terminator:)(rax_22, rax_23, rax_24)
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.11 Print.* Detects calls to `print` preceeded by statements which construct an array of `String`'s.

```
void f(d:)(void* arg1) {
     Type_2 var_48
    Type_3 var_70
    print("Entries:")
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    while (0x1) {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [Int : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary.count.getter(arg1, type metadata for Int, type metadata for Int, protocol witness table for Int))
            float var_a8 = zmm0_1
            int32_t var_204 = 0x1
            print("Average: \(var_a8)")
            return
        }
        int64_t v_1 = v
        int64_t k_1 = k
        print("  \(k_1): \(v_1)")
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            break
        }
        var_48 = v + var_48
    }
    trap(0x6)
}
```

*A.0.12 InfiniteLoopWithFallthrough.* Detects infinite loops with short fallthrough blocks and a `break` then replaces that `break` with the fallthrough

```
void f(d:)(void* arg1) {
    Type_2 var_48
    Type_3 var_70
    print("Entries:")
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type metadata for Int, type metadata for Int, protocol witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    loop {
        option_tup_t var_88
```

```
        &Dictionary.Iterator.next()(&var_88, &var_70, &
     ___swift_instantiateConcreteTypeFromMangledName(&
     demangling cache variable for type metadata for [Int
      : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&
     Dictionary.count.getter(arg1, type metadata for Int,
      type metadata for Int, protocol witness table for
     Int))
            float var_a8 = zmm0_1
            int32_t var_204 = 0x1
            print("Average: \(var_a8)")
            return
        }
        int64_t v_1 = v
        int64_t k_1 = k
        print("  \(k_1): \(v_1)")
        int64_t rax_26
        rax_26.0x0 = add_overflow(v, var_48)
        if (rax_26.0x0 && 0x1 != 0x0) {
            trap(0x6)
        }
        var_48 = v + var_48
    }
}
```

*A.0.13 OverflowCheck.* Detects checks for overflow in arithmetic operations followed by a `trap` operation and removes them. Also performs type propagation on the operands.

```
void f(d:)(void* arg1) {
    int64_t var_48
    Type_3 var_70
    print("Entries:")
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type
     metadata for Int, type metadata for Int, protocol
     witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    loop {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &
     ___swift_instantiateConcreteTypeFromMangledName(&
     demangling cache variable for type metadata for [Int
      : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            float zmm0_1 = float.s(var_48) f/ float.s(&
     Dictionary.count.getter(arg1, type metadata for Int,
      type metadata for Int, protocol witness table for
     Int))
            float var_a8 = zmm0_1
            int32_t var_204 = 0x1
            print("Average: \(var_a8)")
            return
        }
        int64_t v_1 = v
        int64_t k_1 = k
        print("  \(k_1): \(v_1)")
        var_48 = v + var_48
    }
}
```

*A.0.14 InfiniteLoopWithExit.* Detects infinite loops with no fallthrough and one nested block which exits the loop. That block is then made into the fallthrough and converted into a `break` statement within the loop.

```
void f(d:)(void* arg1) {
    int64_t var_48
    Type_22 var_70
    print("Entries:")
    var_48 = 0x0
    void var_38
    &Dictionary.makeIterator()(&var_38, arg1, type
     metadata for Int, type metadata for Int, protocol
     witness table for Int)
    &_memcpy(&var_70, &var_38, 0x28)
    loop {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_70, &
     ___swift_instantiateConcreteTypeFromMangledName(&
     demangling cache variable for type metadata for [Int
      : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            break
        }
        int64_t v_1 = v
        int64_t k_1 = k
        print("  \(k_1): \(v_1)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
     .count.getter(arg1, type metadata for Int, type
     metadata for Int, protocol witness table for Int))
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

*A.0.15 IteratorConstruct.* Detects calls to `makeIterator` and sets the type of the variable. The type information about the iterator's item is gained from the extra parameters passed to `makeIterator`. This type information is also used to set the type of the first variable. This creates the following dynamic rules:

- `IteratorNext`: Detects calls to `Iterator.next` for the iterator variable and sets the output variable to an optional value type of the iterator's item type. This creates the following dynamic rules:
  - `OptionalUnwrap`: Detects when the optional's inner type is copied out and its null field is checked.
- `IteratorLoop`: Detects an an `IteratorConstruct` followed by an infinite loop where the first statement is an `IteratorNext` followed by an `OptionalUnwrap`.

After the original rule is applied:

```
void f(d:)([Int:Int] arg1) {
    int64_t var_48
    Iterator<(Int, Int)> var_38
    print("Entries:")
    var_48 = 0x0
    Iterator<(Int, Int)> var_38 = arg1.makeIterator()
```

```
    loop {
        option_tup_t var_88
        &Dictionary.Iterator.next()(&var_88, &var_38, &
         ___swift_instantiateConcreteTypeFromMangledName(&
         demangling cache variable for type metadata for [Int
          : Int].Iterator))
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            break
        }
        int64_t v_1 = v
        int64_t k_1 = k
        print("  \(k_1): \(v_1)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
     .count.getter(arg1, type metadata for Int, type
     metadata for Int, protocol witness table for Int))
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

After `IteratorNext` is applied:

```
void f(d:)([Int:Int] arg1) {
    int64_t var_48
    Iterator<(Int, Int)> var_38
    print("Entries:")
    var_48 = 0x0
    Iterator<(Int, Int)> var_38 = arg1.makeIterator()
    loop {
        (Int, Int)? var_88
        var_88 = var_38.next()
        int64_t k = var_88.0x0
        int64_t v = var_88.0x8
        int64_t rax_9
        rax_9.0x0 = var_88.0x10
        if (rax_9.0x0 && 0x1 != 0x0) {
            break
        }
        int64_t v_1 = v
        int64_t k_1 = k
        print("  \(k_1): \(v_1)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
     .count.getter(arg1, type metadata for Int, type
     metadata for Int, protocol witness table for Int))
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

After `OptionalUnwrap` is applied:

```
void f(d:)([Int:Int] arg1) {
    Int var_48
    print("Entries:")
    var_48 = 0x0
    Iterator<(Int, Int)> var_38 = arg1.makeIterator()
    loop {
        (Int, Int)? var_88 = var_38.next()
        (Int, Int) (k, v) = var_88
        if (var_88 == nil) {
            break
        }
        print("  \(k): \(v)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
     .count.getter(arg1, type metadata for Int, type
     metadata for Int, protocol witness table for Int))
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

After `IteratorLoop` is applied:

```
void f(d:)([Int:Int] arg1) {
    Int var_48
    print("Entries:")
    var_48 = 0x0
    for (k, v) in arg1 {
        print("  \(k): \(v)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
     .count.getter(arg1, type metadata for Int, type
     metadata for Int, protocol witness table for Int))
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

#### *A.0.16 DictCount.* Detects calls to `Dictionary.count.getter`.

```
void f(d:)([Int:Int] arg1) {
    Int var_48
    print("Entries:")
    var_48 = 0x0
    for (k, v) in arg1 {
        print("  \(k): \(v)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

#### *A.0.17 SwiftVarDecl.* Overrides the `hlil.VarDecl` and `hlil.VarInit` descriptors to display these statements (when assigning a variable with a Swift type) with Swift syntax.

```
void f(d:)([Int:Int] arg1) {
    var var_48: Int
    print("Entries:")
    var_48 = 0x0
    for (k, v) in arg1 {
        print("  \(k): \(v)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

*A.0.18 SwiftFunc.* Overrides the `hlil.Func` descriptor to display the function signature with Swift syntax.

```
func f(arg1: [Int:Int]) -> void {
    var var_48: Int
    print("Entries:")
    var_48 = 0x0
    for (k, v) in arg1 {
        print("  \(k): \(v)")
        var_48 = v + var_48
    }
    float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
    float var_a8 = zmm0_1
    int32_t var_204 = 0x1
    print("Average: \(var_a8)")
    return
}
```

While most of these rules were indeed created specifically for this example, they show how the decompilation can be transformed into almost exactly the original source code using *ideco*.

## B HLIL Descriptors

The following is the inheritance hierarchy of the descriptors used to model HLIL:

- Descriptor
  - DataType
  - Opcode
  - Function
  - Stmt
    - Block
    - If
    - IfElse
    - Switch
    - Case
    - DoWhlie
    - While
    - For
    - Nop
    - Trap
    - NoReturn
    - VarDecl
    - VarInit
    - Assign
    - LabelDecl
    - VoidCall
    - Return
    - Break
    - Continue
  - Expr
    - Int
    - Symbol
      - Data
    - String
    - Var
    - Call
    - Label
    - Goto
    - Jump
    - Intrinsic
    - ExprList
    - UnaryExpr
      - Deref
      - UnaryFunc
    - BinaryExpr
      - BinaryFunc
    - StructField
    - StructFieldDeref
    - ArrayElem

## C Original Crackme

The original Swift code for the crackme. This code was purposely written to use Swift-specific features such as enums, iterators, and closures.

```
import Foundation
import CryptoKit

struct Input {
    static func prompt(_ message: String) -> String {
        print(message, terminator: ": ")
        return readLine() ?? ""
    }
}

enum ValidationResult {
    case success(String)
    case failure(String)
}

struct KeyValidator {
    private let expectedHash: String

    init(expectedHash: String) {
        self.expectedHash = expectedHash
    }

    func validate(_ input: String) -> ValidationResult {
        var inputRev = ""

        for c in input {
            inputRev = "\(c)\(inputRev)"
        }

        let hash = SHA256.hash(data: Data(inputRev.utf8))
    .compactMap {
            String(format: "\%02x", $0)
        }.joined()

        if hash == expectedHash {
            return .success("Access Granted")
        } else {
            return .failure("Invalid key")
        }
    }
}

struct CrackMe {
    private let validator: KeyValidator

    init() {
        let expectedHash = SHA256.hash(data: Data("foobar
    ".utf8)).compactMap {
            String(format: "\%02x", $0)
        }.joined()
        self.validator = KeyValidator(expectedHash:
    expectedHash)
```

```
    }

    func run() -> Bool {
        print("Welcome to the Swift CrackMe!")
        let key = Input.prompt("Enter the secret key")

        switch validator.validate(key) {
        case .success(let message):
            print(message)
            return true
        case .failure(let message):
            print(message)
            return false
        }
    }
}

func main() {
    let challenge = CrackMe()
    if !challenge.run() {
        exit(EXIT_FAILURE)
    }
}

main()
```

# D LLM Inputs

Below are the different versions of the decompilation of the `validate` method given to the different GPT's. The LLM was given the decompilation of the other functions as well but the `validate` function is the most interesting and gave the LLM the most difficulty in understanding. As more rules are introduced, the string interpolation loop in the validate function gets smaller and closer to the original source code.

## D.1 Binary Ninja HLIL

```
int64_t KeyValidator.validate(_:)(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4)
    int512_t zmm0
    zmm0.o = zx.o(0)
    int128_t var_148 = zx.o(0)
    int256_t s_1
    (&__builtin_memset)(s: &s_1, c: 0, n: 0x50)
    int64_t var_f8 = 0
    int128_t s
    (&__builtin_memset)(s: &s, c: 0, n: 0x30)
    void* rax = type metadata accessor for SHA256Digest(0, zmm0)
    void* rax_1 = *(rax - 8)
    int64_t rax_2 = *(rax_1 + 0x40)
    int64_t rdx_1
    int64_t rsi_1
    int64_t rdi_1
    rdx_1, rsi_1, rdi_1 = (&___chkstk_darwin)()
    int64_t var_258
    void* rsp = &var_258 - ((rax_2 + 0xf) & 0xfffffffffffffff0)
    int128_t var_28
    var_28.q = rdi_1
    var_28:8.q = rsi_1
    int128_t var_38
    var_38.q = rdx_1
    var_38:8.q = arg4
    int64_t rax_6
    int64_t rdx_2
    rax_6, rdx_2 = String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0, 1)
    int128_t var_48
    var_48.q = rax_6
    var_48:8.q = rdx_2
    _swift_bridgeObjectRetain(arg2)
    int64_t rax_7
    int64_t rcx_1
    int64_t rdx_3
    int64_t r8_1
    rax_7, rcx_1, rdx_3, r8_1 = String.makeIterator()(arg1, arg2)
    s_1.q = rax_7
    s_1:8.q = rdx_3
    s_1:0x10.q = rcx_1
    s_1:0x18.q = r8_1
    while (true)
    {
        int64_t rax_10
        int64_t rdx_4
        rax_10, rdx_4 = String.Iterator.next()(&s_1)
        if (rdx_4 == 0)
        {
            break
        }
        int128_t var_b8_1
        var_b8_1.q = rax_10
        var_b8_1:8.q = rdx_4
        int64_t rax_13
        int64_t rdx_6
        rax_13, rdx_6 = DefaultStringInterpolati...(literalCapacity:interpolationCount:)(0, 2)
        s.q = rax_13
        s:8.q = rdx_6
        int64_t rax_14
        int64_t rdx_7
        rax_14, rdx_7 = String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0, 1)
        DefaultStringInterpolation.appendLiteral(_:)(rax_14, rdx_7)
        _swift_bridgeObjectRelease(rdx_7)
        int64_t var_d8 = rax_10
        int64_t var_d0_1 = rdx_4
        DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_d8, type metadata for Character, ...)
        int64_t rax_16
        int64_t rdx_9
        rax_16, rdx_9 = String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0, 1)
        DefaultStringInterpolation.appendLiteral(_:)(rax_16, rdx_9)
        _swift_bridgeObjectRelease(rdx_9)
        int64_t rax_17 = var_48.q
        int64_t rdi_9 = var_48:8.q
        _swift_bridgeObjectRetain(rdi_9)
        int64_t var_e8 = rax_17
        int64_t var_e0_1 = rdi_9
        DefaultStringInterpolation.appendInterpolation<A>(_:)(&var_e8, type metadata for String, ...)
        (&outlined destroy of String)(&var_e8)
        int64_t rax_19
        int64_t rdx_11
        rax_19, rdx_11 = String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(" ", 0, 1)
        DefaultStringInterpolation.appendLiteral(_:)(rax_19, rdx_11)
        _swift_bridgeObjectRelease(rdx_11)
        int64_t rax_20 = s.q
```

```
        int64_t rdi_14 = s:8.q
        _swift_bridgeObjectRetain(rdi_14)
        (&outlined destroy of DefaultStringInterpolation)(&s)
        int64_t rax_21
        int64_t rdx_12
        rax_21, rdx_12 = String.init(stringInterpolation:)(rax_20, rdi_14)
        var_48.q = rax_21
        var_48:8.q = rdx_12
        _swift_bridgeObjectRelease(var_48:8.q)
        _swift_bridgeObjectRelease(rdx_4)
    }
    (&outlined destroy of String.Iterator)(&s_1)
    int64_t rax_23 = type metadata accessor for SHA256(0)
    int64_t rax_24
    int64_t rdx_13
    rax_24, rdx_13 = String.utf8.getter(var_48.q, var_48:8.q)
    int64_t var_78 = rax_24
    int64_t var_70 = rdx_13
    int64_t rax_26
    int64_t rdx_15
    rax_26, rdx_15 = Data.init<A>(_:)(&var_78, type metadata for String.UTF8View, ...)
    int64_t var_88 = rax_26
    int64_t var_80 = rdx_15
    int64_t* var_1e0 = &var_88
    static HashFunction.hash<A>(data:)(&var_88, rax_23, type metadata for Data, ...)
    (&outlined destroy of Data)(var_1e0)
    (&___chkstk_darwin)()
    *(rsp - 0x10) = &closure #1 in KeyValidator.validate(_:)
    *(rsp - 8) = 0
    int64_t rax_33 = Sequence.compactMap<A>(_:)(&partial apply for thunk ..., rsp - 0x20, rax, type metadata for String, ...)
    if (var_f8 != 0)
    {
        trap(6)
    }
    (*(rax_1 + 8))(rsp, rax, rax_33, rsp)
    int64_t var_98 = rax_33
    int64_t rax_36 = (&___swift_instantiateConcreteTypeFromMangledName)(&type metadata for [String])
    int64_t rax_37 = (&lazy protocol witness ta...for type [String] and conformance [A])()
    int64_t rax_38
    int64_t rdx_19
    rax_38, rdx_19 = (&default argument 0 of Bi...tionalCollection<>.joined(separator:))(rax_36)
    int64_t rax_39
    int64_t rdx_21
    rax_39, rdx_21 = BidirectionalCollection<>.joined(separator:)(rax_38, rdx_19, rax_36, rax_37)
    _swift_bridgeObjectRelease(rdx_19)
    (&outlined destroy of [String])(&var_98)
    int128_t var_a8
    var_a8.q = rax_39
    var_a8:8.q = rdx_21
    _swift_bridgeObjectRetain(arg4)
    char rax_41 = static String.== infix(_:_:)(rax_39, rdx_21, arg3, arg4)
    _swift_bridgeObjectRelease(arg4)
    int64_t var_238
    char var_221
    int64_t var_230
    if ((rax_41 & 1) != 0)
    {
        int64_t rax_43
        int64_t rdx_24
        rax_43, rdx_24 = String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Access Granted ", 0x12, 0)
        _swift_bridgeObjectRelease(rdx_21)
        (&outlined destroy of String)(&var_48)
        var_238 = rax_43
        var_230 = rdx_24
        var_221 = 0
    }
    else
    {
        int64_t rax_44
        int64_t rdx_26
        rax_44, rdx_26 = String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Invalid key ", 0xf, 0)
        var_258 = rax_44
        _swift_bridgeObjectRelease(rdx_21)
        (&outlined destroy of String)(&var_48)
        var_238 = var_258
        var_230 = rdx_26
        var_221 = 1
    }
    int64_t rcx_13
    rcx_13.b = var_221
    rcx_13.b = rcx_13.b & 1
    return var_238
```

## D.2 With ValueTypes Rules

```
int64_t KeyValidator.validate(_:)(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4) {
        int512_t zmm0
    zmm0.0x0 = 0x0
    int128_t var_148 = 0x0
    int256_t s_1
    &__builtin_memset(&s_1, 0x0, 0x50)
    int64_t var_f8 = 0x0
    Type_3 rax_13
    &__builtin_memset(&rax_13, 0x0, 0x30)
    void* rax = type metadata accessor for SHA256Digest(0x0, zmm0)
    void* rax_1 = *(rax + 0xfffffffffffffff8)
    int64_t rax_2 = *(rax_1 + 0x40)
    int64_t rdx_1
    int64_t rsi_1
    int64_t rdi_1
    rdx_1, rsi_1, rdi_1 = &___chkstk_darwin()
    Type_13 rax_44
    void* rsp = &rax_44 - rax_2 + 0xf && 0xfffffffffffffff0
    int128_t var_28
    var_28.0x0 = rdi_1
    var_28.0x8 = rsi_1
    int128_t var_38
    var_38.0x0 = rdx_1
    var_38.0x8 = arg4
    Type_1 rax_6 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0x0, 0x1)
    Type_1 rax_6
    int64_t rax_7
    int64_t rcx_1
    int64_t rdx_3
    int64_t r8_1
```

```
    rax_7, rcx_1, rdx_3, r8_1 = String.makeIterator()(arg1, arg2)
    s_1.0x0 = rax_7
    s_1.0x8 = rdx_3
    s_1.0x10 = rcx_1
    s_1.0x18 = r8_1
    while (0x1) {
        Type_2 rax_10 = String.Iterator.next()(&s_1)
        if (rdx_4 == 0x0) {
            break
        }
        Type_3 rax_13 = &DefaultStringInterpolation.init(literalCapacity:interpolationCount:)(0x0, 0x2)
        Type_4 rax_14 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_14)
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &rax_10, type metadata for Character, protocol witness table for Character, protocol witness table for Character)
        Type_5 rax_16 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_16)
        &DefaultStringInterpolation.appendInterpolation<A>(_:)(&rax_13, &rax_17, type metadata for String, protocol witness table for String, protocol witness table for String)
        Type_6 rax_19 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0x0, 0x1)
        &DefaultStringInterpolation.appendLiteral(_:)(&rax_13, rax_19)
        Type_7 rax_21 = &String.init(stringInterpolation:)(rax_13)
        rax_6.0x0 = rax_21
    }
    int64_t rax_23 = type metadata accessor for SHA256(0x0)
    Type_8 rax_24 = String.utf8.getter(rax_6.0x0, rax_6.0x8)
    Type_9 rax_26 = Data.init<A>(_:)(&rax_24, type metadata for String.UTF8View, &lazy protocol witness table accessor for type String.UTF8View and conformance String.UTF8View())
    int64_t* var_1e0 = &rax_26
    static HashFunction.hash<A>(data:)(&rax_26, rax_23, type metadata for Data, &lazy protocol witness table accessor for type SHA256 and conformance SHA256(), &lazy protocol witness table accessor for type Data and conformance Data())
    &___chkstk_darwin()
    *(rsp - 0x10) = &closure #1 in KeyValidator.validate(_:)
    *(rsp - 0x8) = 0x0
    int64_t rax_33 = Sequence.compactMap<A>(_:)(&partial apply for thunk for @callee_guaranteed (@unowned UInt8) -> (@owned String?, @error @owned Error), rsp - 0x20, rax, type metadata for String, &lazy protocol witness table accessor for type SHA256Digest and conformance SHA256Digest())
    if (var_f8 != 0x0) {
        trap(0x6)
    }
    *(rax_1 + 0x8)(rsp, rax, rax_33, rsp)
    int64_t var_98 = rax_33
    int64_t rax_36 = &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [String])
    int64_t rax_37 = &lazy protocol witness table accessor for type [String] and conformance [A]()
    Type_10 rax_38 = &default argument 0 of BidirectionalCollection<>.joined(separator:)(rax_36)
    Type_11 rax_39 = BidirectionalCollection<>.joined(separator:)(rax_38, rax_36, rax_37)
    int128_t var_a8
    var_a8.0x0 = rax_39
    char rax_41 = static String.== infix(_:_:)(rax_39, arg3, arg4)
    Type_12 rax_43
    char var_221
    if (rax_41 && 0x1 != 0x0) {
        Type_13 rax_44 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Access Granted ", 0x12, 0x0)
        var_221 = 0x0
    }
    else {
        Type_13 rax_44 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Invalid key ", 0xf, 0x0)
        var_221 = 0x1
    }
    int64_t rcx_13
    rcx_13.0x0 = var_221
    rcx_13.0x0 = rcx_13.0x0 && 0x1
    return rax_43
}
```

## D.3 With ValueTypes and Swift Rules

This version of the decompilation uses many of the rules from Appendix A to recover the string interpolation.

```
func KeyValidator.validate(arg1: int64_t, arg2: int64_t, arg3: int64_t, arg4: int64_t) -> int64_t {
        int512_t zmm0
    zmm0.0x0 = 0x0
    int128_t var_148 = 0x0
    int256_t s_1
    &__builtin_memset(&s_1, 0x0, 0x50)
    int64_t var_f8 = 0x0
    Type_3 rax_13
    &__builtin_memset(&rax_13, 0x0, 0x30)
    void* rax = type metadata accessor for SHA256Digest(0x0, zmm0)
    void* rax_1 = *(rax + 0xfffffffffffffff8)
    int64_t rax_2 = *(rax_1 + 0x40)
    int64_t rdx_1
    int64_t rsi_1
    int64_t rdi_1
    rdx_1, rsi_1, rdi_1 = &___chkstk_darwin()
    Type_13 rax_44
    void* rsp = &rax_44 - rax_2 + 0xf && 0xfffffffffffffff0
    int128_t var_28
    var_28.0x0 = rdi_1
    var_28.0x8 = rsi_1
    int128_t var_38
    var_38.0x0 = rdx_1
    var_38.0x8 = arg4
    var rax_6: String
    for rax_10 in arg1 {
        Type_1 rax_17 = rax_6
```

```
        rax_6 = "\(rax_10)\(rax_17)"
    }
    int64_t rax_23 = type metadata accessor for SHA256(0x0)
    Type_8 rax_24 = String.utf8.getter(rax_6.0x0, rax_6.0x8)
    Type_9 rax_26 = Data.init<A>(_:)(&rax_24, type metadata for String.UTF8View, &lazy protocol witness table accessor for type String.UTF8View and conformance String.UTF8View())
    int64_t* var_1e0 = &rax_26
    static HashFunction.hash<A>(data:)(&rax_26, rax_23, type metadata for Data, &lazy protocol witness table accessor for type SHA256 and conformance SHA256(), &lazy protocol witness table accessor for type Data and conformance Data())
    &___chkstk_darwin()
    *(rsp - 0x10) = &closure #1 in KeyValidator.validate(_:)
    *(rsp - 0x8) = 0x0
    int64_t rax_33 = Sequence.compactMap<A>(_:)(&partial apply for thunk for @callee_guaranteed (@unowned UInt8) -> (@owned String?, @error @owned Error), rsp - 0x20, rax, type metadata for String, &lazy protocol witness table accessor for type SHA256Digest and conformance SHA256Digest())
    if (var_f8 != 0x0) {
        trap(0x6)
    }
    *(rax_1 + 0x8)(rsp, rax, rax_33, rsp)
    int64_t var_98 = rax_33
    int64_t rax_36 = &___swift_instantiateConcreteTypeFromMangledName(&demangling cache variable for type metadata for [String])
    int64_t rax_37 = &lazy protocol witness table accessor for type [String] and conformance [A]()
    Type_10 rax_38 = &default argument 0 of BidirectionalCollection<>.joined(separator:)(rax_36)
    Type_11 rax_39 = BidirectionalCollection<>.joined(separator:)(rax_38, rax_36, rax_37)
    int128_t var_a8
    var_a8.0x0 = rax_39
    char rax_41 = static String.== infix(_:_:)(rax_39, arg3, arg4)
    Type_12 rax_43
    char var_221
    if (rax_41 && 0x1 != 0x0) {
        Type_13 rax_44 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Access Granted ", 0x12, 0x0)
        var_221 = 0x0
    }
    else {
        Type_13 rax_44 = &String.init(_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("Invalid key ", 0xf, 0x0)
        var_221 = 0x1
    }
    int64_t rcx_13
    rcx_13.0x0 = var_221
    rcx_13.0x0 = rcx_13.0x0 && 0x1
    return rax_43
}
```