

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>

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SURE '25, Taipei, Taiwan

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1910-3/25/10

<https://doi.org/10.1145/3733822.3764668>

```
1 func f(d: [Int: Int]) {
2     print("Entries:")
3     var sum = 0
4
5     for (k, v) in d {
6         print("  \k): \v)")
7         sum += v
8     }
9
10    let avg = Float(sum) / Float(d.count)
11    print("Average: \avg)")
12 }
13
```

(a) Original Swift Code

```
1 func f(arg1: [Int: Int]) -> void {
2     var var_48: Int
3     print("Entries:")
4     var_48 = 0x0
5     for (k, v) in arg1 {
6         print("  \k): \v)")
7         var_48 = v + var_48
8     }
9     float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
10    float var_a8 = zmm0_1
11    int32_t var_204 = 0x1
12    print("Average: \var_a8)")
13    return
14 }
15
```

(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.

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.

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

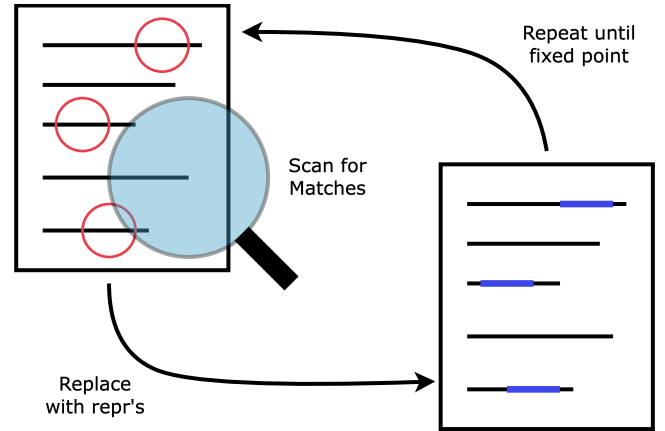


Figure 2: The annotated decompilation is scanned for matching templates and the found candidates are substituted. This process is iterated until no new matches are found.

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.

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 sub-tree in the CST is deemed to be "hidden" and will not be displayed in the rendered text (as in Figure 4).

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`).

```
1 class Call(Expr):
2     func: Expr
3     args: list[Expr]
4
5     def __repr__(self):
6         return eval_repr('$func(${args*<, >})')
7
8 class VarInit(Stmt):
9     lhs: Var
10    rhs: Expr
11
12    def __repr__(self):
13        return eval_repr('${lhs.data_type} $lhs = $rhs')
```

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

```
1 import hlil
2
3 class ReferenceCountingOp(hlil.Call):
4     @staticmethod
5     def match():
6         (ok, vars) = eval_tmpl('$func(${args*<, >})')
7
8         return ok and vars['func'].name in {
9             '_objc_retain', '_objc_release',
10             '_swift_retain', '_swift_release',
11             ...
12         }
13
14    def __repr__(self):
15        return eval_repr('${args[0]}')
```

```
17 class VoidOp(hlil.Stmt):
18    call: ReferenceCountingOp
19
20    @staticmethod
21    def match():
22        return eval_tmpl('$call')[0]
23
24    def __repr__(self):
25        return eval_repr('')
```

```
27 class OpWithValue(hlil.VarInit):
28    dt: hlil.DataType
29    rhs: ReferenceCountingOp
30
31    @staticmethod
32    def match():
33        return eval_tmpl('$dt $lhs = $rhs')[0]
34
35    def __init__(self):
36        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.

4.2 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

```

1 int64_t a = 1; 1 int64_t a = 1;
2 _objc_release(b); 1 int64_t a = 1;
3 int64_t c = 2; 3 int64_t c = 2; 2 int64_t c = 2;

```

(a) Original (b) After matching (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

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

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

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

```

1 int64_t rax_13
2 int64_t rdx_6
3 rax_13, rdx_6 = DefaultStringInterpolation.init(0, 2)
4 s.q = rax_13
5 s:8.q = rdx_6

```

(a) Original Decompilation

```

1 ValueType rax_13 = DefaultStringInterpolation.init(0, 2)
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.

```

1 class ValueTypeInit(hlil.VarInit):
2     dt1: hlil.DataType
3     dt2: hlil.DataType
4     var2: hlil.Var
5     rhs: hlil.Call
6
7     @staticmethod
8     def match():
9         return eval_tmpl('
10             $dt1 $lhs
11             $dt2 $var2
12             $lhs, $var2 = $rhs
13         ')[0]
14
15     def __init__(self):
16         dt1 = self.lhs.data_type
17         dt2 = self.var2.data_type
18
19         dt = create_type('ValueType')
20         self.lhs.set_type(dt)
21
22     def match_copy():
23         (ok, vars) = eval_tmpl('
24             $dst1 = $field1
25             $dst2 = $field2
26         ')
27         return ok and \
28             vars['field1'].data_type == dt and \
29             vars['field2'].data_type == dt2
30
31     def init_copy(self):
32         self.dst1.set_type(dt)
33         self.dst2.set_type(dt2)
34
35     create_descriptor(
36         name=f'{dt}_Copy',
37         match=match_copy,
38         init=init_copy,
39         repr=lambda: eval_repr('$dst1 = $field1'),
40         annos={'dst1':hlil.Expr, 'field1':dt1, ...},
41     )

```

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 ("rabooof"), 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 Leonahenah 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:

```

1 int64_t f(d:)(void* arg1) {
2     void* var_40
3     &memset(&var_40, 0x0, 0x8)
4     int64_t var_48
5     &memset(&var_48, 0x0, 0x8)
6     void var_70
7     &memset(&var_70, 0x0, 0x28)
8     float var_90
9     &memset(&var_90, 0x0, 0x4)
10    int64_t var_a0
11    &memset(&var_a0, 0x0, 0x10)
12    int64_t k_2
13    &memset(&k_2, 0x0, 0x8)
14    int64_t v_2
15    &memset(&v_2, 0x0, 0x8)
16    int64_t var_c8
17    &memset(&var_c8, 0x0, 0x10)
18    var_40 = arg1
19    int64_t rax
20    int64_t* rdx
21    rax, rdx = &_allocateUninitializedArray<A>(_:)(0x1,
22    type metadata for Any + 0x8)
23    int64_t rax_1
24    int64_t rdx_1
25    rax_1, rdx_1 = &String.init(_builtinStringLiteral:
26    utf8CodeUnitCount:isASCII:)(“Entries:”, 0x8, 0x1)
27    rdx[0x3] = type metadata for String
28    *(rdx) = rax_1
29    rdx[0x1] = rdx_1
30    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
31    rax)
32    int64_t rax_4
33    int64_t rdx_3
34    rax_4, rdx_3 = &default argument 1 of print(_:
35    separator:terminator:>()
36    int64_t rax_5
37    int64_t rdx_4

```

```

34    rax_5, rdx_4 = &default argument 2 of print(_:
35    separator:terminator:>()
36    &print(_:separator:terminator:)(rax_3, rax_4, rdx_3,
37    rax_5, rdx_4)
38    &_swift_bridgeObjectRelease(rdx_4)
39    &_swift_bridgeObjectRelease(rdx_3)
40    &_swift_bridgeObjectRelease(rax_3)
41    var_48 = 0x0
42    &_swift_bridgeObjectRetain(arg1)
43    void var_38
44    &Dictionary.makeIterator()(&var_38, arg1, type
45    metadata for Int, type metadata for Int, protocol
46    witness table for Int)
47    &memcpy(&var_70, &var_38, 0x28)
48    while (0x1) {
49        option_tup_t var_88
50        &Dictionary.Iterator.next()(&var_88, &var_70, &
51        __swift_instantiateConcreteTypeFromMangledName(&
52        demangling cache variable for type metadata for [Int
53        : Int].Iterator))
54        int64_t k = var_88.0x0
55        int64_t v = var_88.0x8
56        int64_t rax_9
57        rax_9.0x0 = var_88.0x10
58        if (rax_9.0x0 && 0x1 != 0x0) {
59            &outlined destroy of [Int : Int].Iterator(&
60            var_70)
61            float zmm0_1 = float.s(var_48) f/ float.s(&
62            Dictionary.count.getter(arg1, type metadata for Int,
63            type metadata for Int, protocol witness table for
64            Int))
65            var_90 = zmm0_1
66            int64_t rax_29
67            int64_t* rdx_20
68            rax_29, rdx_20 = &_allocateUninitializedArray
69            <A>(_:)(0x1)
70            int64_t rax_30
71            int64_t rdx_21
72            rax_30, rdx_21 = &DefaultStringInterpolation.
73            init(literalCapacity:interpolationCount:)(0x9, 0x1)
74            var_a0 = rax_30
75            int32_t var_204 = 0x1
76            void* rax_31
77            void* rdx_22
78            rax_31, rdx_22 = &String.init(
79            _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(“
80            Average: ”, 0x9, 0x1)
81            &DefaultStringInterpolation.appendLiteral(_:)(
82            &var_a0, rax_31, rdx_22)
83            &_swift_bridgeObjectRelease(rdx_22)
84            float var_a8 = zmm0_1
85            &DefaultStringInterpolation.
86            appendInterpolation<A>(_:)(&var_a0, &var_a8, type
87            metadata for Float, protocol witness table for Float
88            , protocol witness table for Float)
89            void* rax_32
90            void* rdx_24
91            rax_32, rdx_24 = &String.init(
92            _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
93            data_100003f7f, 0x0, 0x1)
94            &DefaultStringInterpolation.appendLiteral(_:)(
95            &var_a0, rax_32, rdx_24)
96            &_swift_bridgeObjectRelease(rdx_24)
97            int64_t rax_33 = var_a0
98            &_swift_bridgeObjectRetain(rdx_21)
99            &outlined destroy of
100            DefaultStringInterpolation(&var_a0)
101            int64_t rax_34
102            int64_t rdx_25

```

```

80         rax_34, rdx_25 = &String.init(
            stringInterpolation:)(rax_33, rdx_21)
81         rdx_20[0x3] = type metadata for String
82         *(rdx_20) = rax_34
83         rdx_20[0x1] = rdx_25
84         int64_t rax_36 = &_finalizeUninitializedArray
            <A>(_:)(rax_29)
85         int64_t rax_37
86         int64_t rdx_27
87         rax_37, rdx_27 = &default argument 1 of print
            (_:separator:terminator:>()
88         int64_t rax_38
89         int64_t rdx_28
90         rax_38, rdx_28 = &default argument 2 of print
            (_:separator:terminator:>()
91         &print(_:separator:terminator:)(rax_36,
            rax_37, rdx_27, rax_38, rdx_28)
92         &_swift_bridgeObjectRelease(rdx_28)
93         &_swift_bridgeObjectRelease(rdx_27)
94         return &_swift_bridgeObjectRelease(rax_36)
95     }
96     k_2 = k
97     v_2 = v
98     int64_t rax_12
99     int64_t* rdx_7
100    rax_12, rdx_7 = &_allocateUninitializedArray<A>(_
        :)(0x1, type metadata for Any + 0x8)
101    int64_t rax_13
102    int64_t rdx_8
103    rax_13, rdx_8 = &DefaultStringInterpolation.init(
        literalCapacity:interpolationCount:)(0x4, 0x2)
104    var_c8 = rax_13
105    void* rax_14
106    void* rdx_9
107    rax_14, rdx_9 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
        ", 0x2, 0x1)
108    &DefaultStringInterpolation.appendLiteral(_:)(&
        var_c8, rax_14, rdx_9)
109    &_swift_bridgeObjectRelease(rdx_9)
110    int64_t k_1 = k
111    &DefaultStringInterpolation.appendInterpolation<A
        >(_:)(&var_c8, &k_1, type metadata for Int, protocol
        witness table for Int)
112    void* rax_16
113    void* rdx_11
114    rax_16, rdx_11 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
        ", 0x2, 0x1)
115    &DefaultStringInterpolation.appendLiteral(_:)(&
        var_c8, rax_16, rdx_11)
116    &_swift_bridgeObjectRelease(rdx_11)
117    int64_t v_1 = v
118    &DefaultStringInterpolation.appendInterpolation<A
        >(_:)(&var_c8, &v_1, type metadata for Int, protocol
        witness table for Int)
119    void* rax_18
120    void* rdx_13
121    rax_18, rdx_13 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
        data_100003f7f, 0x0, 0x1)
122    &DefaultStringInterpolation.appendLiteral(_:)(&
        var_c8, rax_18, rdx_13)
123    &_swift_bridgeObjectRelease(rdx_13)
124    int64_t rax_19 = var_c8
125    &_swift_bridgeObjectRetain(rdx_8)
126    &outlined destroy of DefaultStringInterpolation(&
        var_c8)
127    int64_t rax_20

```

```

128    int64_t rdx_14
129    rax_20, rdx_14 = &String.init(stringInterpolation
        :)(rax_19, rdx_8)
130    rdx_7[0x3] = type metadata for String
131    *(rdx_7) = rax_20
132    rdx_7[0x1] = rdx_14
133    int64_t rax_22 = &_finalizeUninitializedArray<A>(_
        :)(rax_12)
134    int64_t rax_23
135    int64_t rdx_16
136    rax_23, rdx_16 = &default argument 1 of print(_:
        separator:terminator:>()
137    int64_t rax_24
138    int64_t rdx_17
139    rax_24, rdx_17 = &default argument 2 of print(_:
        separator:terminator:>()
140    &print(_:separator:terminator:)(rax_22, rax_23,
        rdx_16, rax_24, rdx_17)
141    &_swift_bridgeObjectRelease(rdx_17)
142    &_swift_bridgeObjectRelease(rdx_16)
143    &_swift_bridgeObjectRelease(rax_22)
144    int64_t rax_26
145    rax_26.0x0 = add_overflow(v, var_48)
146    if (rax_26.0x0 && 0x1 != 0x0) {
147        break
148    }
149    var_48 = v + var_48
150 }
151 trap(0x6)
152 }

```

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.

```

1 int64_t f(d:)(void* arg1) {
2     Type_2 var_48
3     Type_3 var_70
4     Type_5 var_a0
5     Type_8 var_c8
6     int64_t rax
7     int64_t* rdx
8     rax, rdx = &_allocateUninitializedArray<A>(_:)(0x1,
        type metadata for Any + 0x8)
9     int64_t rax_1
10    int64_t rdx_1
11    rax_1, rdx_1 = &String.init(_builtinStringLiteral:
        utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
12    rdx[0x3] = type metadata for String
13    *(rdx) = rax_1
14    rdx[0x1] = rdx_1
15    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
        rax)
16    int64_t rax_4
17    int64_t rdx_3

```



```

18     rax_4, rdx_3 = &default argument 1 of print(_:
19         separator:terminator:~)()
20     int64_t rax_5
21     int64_t rdx_4
22     rax_5, rdx_4 = &default argument 2 of print(_:
23         separator:terminator:~)()
24     &print(_:separator:terminator:~)(rax_3, rax_4, rdx_3,
25         rax_5, rdx_4)
26     &_swift_bridgeObjectRelease(rdx_4)
27     &_swift_bridgeObjectRelease(rdx_3)
28     &_swift_bridgeObjectRelease(rax_3)
29     var_48 = 0x0
30     &_swift_bridgeObjectRetain(arg1)
31     void var_38
32     &Dictionary.makeIterator()(&var_38, arg1, type
33         metadata for Int, type metadata for Int, protocol
34         witness table for Int)
35     &memcpy(&var_70, &var_38, 0x28)
36     while (0x1) {
37         option_tup_t var_88
38         &Dictionary.Iterator.next()(&var_88, &var_70, &
39             __swift_instantiateConcreteTypeFromMangledName(&
40                 demangling cache variable for type metadata for [Int
41                 : Int].Iterator))
42         int64_t k = var_88.0x0
43         int64_t v = var_88.0x8
44         int64_t rax_9
45         rax_9.0x0 = var_88.0x10
46         if (rax_9.0x0 && 0x1 != 0x0) {
47             &outlined destroy of [Int : Int].Iterator(&
48                 var_70)
49             float zmm0_1 = float.s(var_48) f/ float.s(&
50                 Dictionary.count.getter(arg1, type metadata for Int,
51                 type metadata for Int, protocol witness table for
52                 Int))
53             int64_t rax_29
54             int64_t* rdx_20
55             rax_29, rdx_20 = &_allocateUninitializedArray
56                 <A>(_:)(0x1)
57             int64_t rax_30
58             int64_t rdx_21
59             rax_30, rdx_21 = &DefaultStringInterpolation.
60                 init(literalCapacity:interpolationCount:~)(0x9, 0x1)
61             var_a0 = rax_30
62             int32_t var_204 = 0x1
63             void* rax_31
64             void* rdx_22
65             rax_31, rdx_22 = &String.init(
66                 _builtinStringLiteral:utf8CodeUnitCount:isASCII:~)("
67                 Average: ", 0x9, 0x1)
68             &DefaultStringInterpolation.appendLiteral(_:)(
69                 &var_a0, rax_31, rdx_22)
70             &_swift_bridgeObjectRelease(rdx_22)
71             float var_a8 = zmm0_1
72             &DefaultStringInterpolation.
73                 appendInterpolation<A>(_:)(&var_a0, &var_a8, type
74                 metadata for Float, protocol witness table for Float
75                 , protocol witness table for Float)
76             void* rax_32
77             void* rdx_24
78             rax_32, rdx_24 = &String.init(
79                 _builtinStringLiteral:utf8CodeUnitCount:isASCII:~)(&
80                 data_100003f7f, 0x0, 0x1)
81             &DefaultStringInterpolation.appendLiteral(_:)(
82                 &var_a0, rax_32, rdx_24)
83             &_swift_bridgeObjectRelease(rdx_24)
84             int64_t rax_33 = var_a0
85             &_swift_bridgeObjectRetain(rdx_21)

```

```

63         &outlined destroy of
64         DefaultStringInterpolation(&var_a0)
65         int64_t rax_34
66         int64_t rdx_25
67         rax_34, rdx_25 = &String.init(
68             stringInterpolation:~)(rax_33, rdx_21)
69         rdx_20[0x3] = type metadata for String
70         *(rdx_20) = rax_34
71         rdx_20[0x1] = rdx_25
72         int64_t rax_36 = &_finalizeUninitializedArray
73             <A>(_:)(rax_29)
74         int64_t rax_37
75         int64_t rdx_27
76         rax_37, rdx_27 = &default argument 1 of print
77             (_:separator:terminator:~)()
78         int64_t rax_38
79         int64_t rdx_28
80         rax_38, rdx_28 = &default argument 2 of print
81             (_:separator:terminator:~)()
82         &print(_:separator:terminator:~)(rax_36,
83             rax_37, rdx_27, rax_38, rdx_28)
84         &_swift_bridgeObjectRelease(rdx_28)
85         &_swift_bridgeObjectRelease(rdx_27)
86         return &_swift_bridgeObjectRelease(rax_36)
87     }
88     int64_t rax_12
89     int64_t* rdx_7
90     rax_12, rdx_7 = &_allocateUninitializedArray<A>(_:)(0x1, type
91         metadata for Any + 0x8)
92     int64_t rax_13
93     int64_t rdx_8
94     rax_13, rdx_8 = &DefaultStringInterpolation.init(
95         literalCapacity:interpolationCount:~)(0x4, 0x2)
96     var_c8 = rax_13
97     void* rax_14
98     void* rdx_9
99     rax_14, rdx_9 = &String.init(
100         _builtinStringLiteral:utf8CodeUnitCount:isASCII:~)("
101         ", 0x2, 0x1)
102     &DefaultStringInterpolation.appendLiteral(_:)(&
103         var_c8, rax_14, rdx_9)
104     &_swift_bridgeObjectRelease(rdx_9)
105     int64_t k_1 = k
106     &DefaultStringInterpolation.appendInterpolation<A>
107         >(_:)(&var_c8, &k_1, type metadata for Int, protocol
108         witness table for Int)
109     void* rax_16
110     void* rdx_11
111     rax_16, rdx_11 = &String.init(
112         _builtinStringLiteral:utf8CodeUnitCount:isASCII:~)(":
113         ", 0x2, 0x1)
114     &DefaultStringInterpolation.appendLiteral(_:)(&
115         var_c8, rax_16, rdx_11)
116     &_swift_bridgeObjectRelease(rdx_11)
117     int64_t v_1 = v
118     &DefaultStringInterpolation.appendInterpolation<A>
119         >(_:)(&var_c8, &v_1, type metadata for Int, protocol
120         witness table for Int)
121     void* rax_18
122     void* rdx_13
123     rax_18, rdx_13 = &String.init(
124         _builtinStringLiteral:utf8CodeUnitCount:isASCII:~)(&
125         data_100003f7f, 0x0, 0x1)
126     &DefaultStringInterpolation.appendLiteral(_:)(&
127         var_c8, rax_18, rdx_13)
128     &_swift_bridgeObjectRelease(rdx_13)
129     int64_t rax_19 = var_c8
130     &_swift_bridgeObjectRetain(rdx_8)

```

```

110      &outlined destroy of DefaultStringInterpolation(&
      var_c8)
111      int64_t rax_20
112      int64_t rdx_14
113      rax_20, rdx_14 = &String.init(stringInterpolation
      :)(rax_19, rdx_8)
114      rdx_7[0x3] = type metadata for String
115      *(rdx_7) = rax_20
116      rdx_7[0x1] = rdx_14
117      int64_t rax_22 = &_finalizeUninitializedArray<A>(&
      _:)(rax_12)
118      int64_t rax_23
119      int64_t rdx_16
120      rax_23, rdx_16 = &default argument 1 of print(_:
      separator:terminator:>()
121      int64_t rax_24
122      int64_t rdx_17
123      rax_24, rdx_17 = &default argument 2 of print(_:
      separator:terminator:>()
124      &print(_:separator:terminator:)(rax_22, rax_23,
      rdx_16, rax_24, rdx_17)
125      &_swift_bridgeObjectRelease(rdx_17)
126      &_swift_bridgeObjectRelease(rdx_16)
127      &_swift_bridgeObjectRelease(rax_22)
128      int64_t rax_26
129      rax_26.0x0 = add_overflow(v, var_48)
130      if (rax_26.0x0 && 0x1 != 0x0) {
131          break
132      }
133      var_48 = v + var_48
134      }
135      trap(0x6)
136      }

```

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.

```

1  int64_t f(d:)(void* arg1) {
2      Type_2 var_48
3      Type_3 var_70
4      Type_14 rax_30
5      Type_21 rax_13
6      Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1,
      type metadata for Any + 0x8)
7      Type_10 rax_1 = &String.init(_builtinStringLiteral:
      utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
8      rdx[0x3] = type metadata for String
9      *(rdx) = rax_1
10     int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
      rax)
11     Type_11 rax_4 = &default argument 1 of print(_:
      separator:terminator:>()
12     Type_12 rax_5 = &default argument 2 of print(_:
      separator:terminator:>()
13     &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
14     &_swift_bridgeObjectRelease(rdx_4)
15     &_swift_bridgeObjectRelease(rdx_3)
16     &_swift_bridgeObjectRelease(rax_3)
17     var_48 = 0x0

```

```

18     &_swift_bridgeObjectRetain(arg1)
19     void var_38
20     &Dictionary.makeIterator()(&var_38, arg1, type
      metadata for Int, type metadata for Int, protocol
      witness table for Int)
21     &memcpy(&var_70, &var_38, 0x28)
22     while (0x1) {
23         option_tup_t var_88
24         &Dictionary.Iterator.next()(&var_88, &var_70, &
      __swift_instantiateConcreteTypeFromMangledName(&
      demangling cache variable for type metadata for [Int
      : Int].Iterator))
25         int64_t k = var_88.0x0
26         int64_t v = var_88.0x8
27         int64_t rax_9
28         rax_9.0x0 = var_88.0x10
29         if (rax_9.0x0 && 0x1 != 0x0) {
30             &outlined destroy of [Int : Int].Iterator(&
      var_70)
31             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))
32             Type_13 rax_29 = &_allocateUninitializedArray
      <A>(_:)(0x1)
33             Type_14 rax_30 = &DefaultStringInterpolation.
      init(literalCapacity:interpolationCount:)(0x9, 0x1)
34             int32_t var_204 = 0x1
35             Type_15 rax_31 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
      "Average: ", 0x9, 0x1)
36             &DefaultStringInterpolation.appendLiteral(_:)(
      &rax_30, rax_31)
37             &_swift_bridgeObjectRelease(rdx_22)
38             float var_a8 = zmm0_1
39             &DefaultStringInterpolation.
      appendInterpolation<A>(_:)(&rax_30, &var_a8, type
      metadata for Float, protocol witness table for Float
      , protocol witness table for Float)
40             Type_16 rax_32 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
      data_100003f7f, 0x0, 0x1)
41             &DefaultStringInterpolation.appendLiteral(_:)(
      &rax_30, rax_32)
42             &_swift_bridgeObjectRelease(rdx_24)
43             &_swift_bridgeObjectRetain(rdx_21)
44             &outlined destroy of
      DefaultStringInterpolation(&rax_30)
45             Type_17 rax_34 = &String.init(
      stringInterpolation:)(rax_30)
46             rdx_20[0x3] = type metadata for String
47             *(rdx_20) = rax_34
48             int64_t rax_36 = &_finalizeUninitializedArray
      <A>(_:)(rax_29)
49             Type_18 rax_37 = &default argument 1 of print
      (_:separator:terminator:>()
50             Type_19 rax_38 = &default argument 2 of print
      (_:separator:terminator:>()
51             &print(_:separator:terminator:)(rax_36,
      rax_37, rax_38)
52             &_swift_bridgeObjectRelease(rdx_28)
53             &_swift_bridgeObjectRelease(rdx_27)
54             return &_swift_bridgeObjectRelease(rax_36)
55         }
56         Type_20 rax_12 = &_allocateUninitializedArray<A>(&
      _:)(0x1, type metadata for Any + 0x8)
57         Type_21 rax_13 = &DefaultStringInterpolation.init
      (literalCapacity:interpolationCount:)(0x4, 0x2)

```

```

58     Type_22 rax_14 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
        ", 0x2, 0x1)
59     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_14)
60     &_swift_bridgeObjectRelease(rdx_9)
61     int64_t k_1 = k
62     &DefaultStringInterpolation.appendInterpolation<A>
        >(_:)(&rax_13, &k_1, type metadata for Int, protocol
        witness table for Int)
63     Type_23 rax_16 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
        ", 0x2, 0x1)
64     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_16)
65     &_swift_bridgeObjectRelease(rdx_11)
66     int64_t v_1 = v
67     &DefaultStringInterpolation.appendInterpolation<A>
        >(_:)(&rax_13, &v_1, type metadata for Int, protocol
        witness table for Int)
68     Type_24 rax_18 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
        data_100003f7f, 0x0, 0x1)
69     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_18)
70     &_swift_bridgeObjectRelease(rdx_13)
71     &_swift_bridgeObjectRetain(rdx_8)
72     &outlined destroy of DefaultStringInterpolation(&
        rax_13)
73     Type_25 rax_20 = &String.init(stringInterpolation
        :)(rax_13)
74     rdx_7[0x3] = type metadata for String
75     *(rdx_7) = rax_20
76     int64_t rax_22 = &_finalizeUninitializedArray<A>(&
        _:)(rax_12)
77     Type_26 rax_23 = &default argument 1 of print(_:
        separator:terminator:>()
78     Type_27 rax_24 = &default argument 2 of print(_:
        separator:terminator:>()
79     &print(_:separator:terminator:)(rax_22, rax_23,
        rax_24)
80     &_swift_bridgeObjectRelease(rdx_17)
81     &_swift_bridgeObjectRelease(rdx_16)
82     &_swift_bridgeObjectRelease(rax_22)
83     int64_t rax_26
84     rax_26.0x0 = add_overflow(v, var_48)
85     if (rax_26.0x0 && 0x1 != 0x0) {
86         break
87     }
88     var_48 = v + var_48
89 }
90 trap(0x6)
91 }

```

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

```

1 void f(d:)(void* arg1) {
2     Type_21 var_48
3     Type_22 var_70
4     Type_6 rax_30
5     Type_13 rax_13
6     Type_1 rax = &_allocateUninitializedArray<A>(_:)(0x1,
        type metadata for Any + 0x8)
7     Type_2 rax_1 = &String.init(_builtinStringLiteral:
        utf8CodeUnitCount:isASCII:)("Entries:", 0x8, 0x1)
8     rdx[0x3] = type metadata for String
9     *(rdx) = rax_1

```

```

10     int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
        rax)
11     Type_3 rax_4 = &default argument 1 of print(_:
        separator:terminator:>()
12     Type_4 rax_5 = &default argument 2 of print(_:
        separator:terminator:>()
13     &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
14     var_48 = 0x0
15     void var_38
16     &Dictionary.makeIterator()(&var_38, arg1, type
        metadata for Int, type metadata for Int, protocol
        witness table for Int)
17     &memcpy(&var_70, &var_38, 0x28)
18     while (0x1) {
19         option_tup_t var_88
20         &Dictionary.Iterator.next()(&var_88, &var_70, &
        __swift_instantiateConcreteTypeFromMangledName(&
        demangling cache variable for type metadata for [Int
        : Int].Iterator))
21         int64_t k = var_88.0x0
22         int64_t v = var_88.0x8
23         int64_t rax_9
24         rax_9.0x0 = var_88.0x10
25         if (rax_9.0x0 && 0x1 != 0x0) {
26             &outlined destroy of [Int : Int].Iterator(&
        var_70)
27             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))
28             Type_5 rax_29 = &_allocateUninitializedArray<
        A>(_:)(0x1)
29             Type_6 rax_30 = &DefaultStringInterpolation.
        init(literalCapacity:interpolationCount:)(0x9, 0x1)
30             int32_t var_204 = 0x1
31             Type_7 rax_31 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
        Average: ", 0x9, 0x1)
32             &DefaultStringInterpolation.appendLiteral(_:)(
        &rax_30, rax_31)
33             float var_a8 = zmm0_1
34             &DefaultStringInterpolation.
        appendInterpolation<A>(_:)(&rax_30, &var_a8, type
        metadata for Float, protocol witness table for Float
        , protocol witness table for Float)
35             Type_8 rax_32 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
        data_100003f7f, 0x0, 0x1)
36             &DefaultStringInterpolation.appendLiteral(_:)(
        &rax_30, rax_32)
37             &outlined destroy of
        DefaultStringInterpolation(&rax_30)
38             Type_9 rax_34 = &String.init(
        stringInterpolation:)(rax_30)
39             rdx_20[0x3] = type metadata for String
40             *(rdx_20) = rax_34
41             int64_t rax_36 = &_finalizeUninitializedArray
        <A>(_:)(rax_29)
42             Type_10 rax_37 = &default argument 1 of print
        (_:separator:terminator:>()
43             Type_11 rax_38 = &default argument 2 of print
        (_:separator:terminator:>()
44             &print(_:separator:terminator:)(rax_36,
        rax_37, rax_38)
45             return
46         }
47         Type_12 rax_12 = &_allocateUninitializedArray<A>(&
        _:)(0x1, type metadata for Any + 0x8)

```

```

48     Type_13 rax_13 = &DefaultStringInterpolation.init
      (literalCapacity:interpolationCount:)(0x4, 0x2)
49     Type_14 rax_14 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
      ", 0x2, 0x1)
50     &DefaultStringInterpolation.appendLiteral(_:)(&
      rax_13, rax_14)
51     int64_t k_1 = k
52     &DefaultStringInterpolation.appendInterpolation<A
      >(_:)(&rax_13, &k_1, type metadata for Int, protocol
      witness table for Int)
53     Type_15 rax_16 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
      ", 0x2, 0x1)
54     &DefaultStringInterpolation.appendLiteral(_:)(&
      rax_13, rax_16)
55     int64_t v_1 = v
56     &DefaultStringInterpolation.appendInterpolation<A
      >(_:)(&rax_13, &v_1, type metadata for Int, protocol
      witness table for Int)
57     Type_16 rax_18 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
      data_100003f7f, 0x0, 0x1)
58     &DefaultStringInterpolation.appendLiteral(_:)(&
      rax_13, rax_18)
59     &outlined destroy of DefaultStringInterpolation(&
      rax_13)
60     Type_17 rax_20 = &String.init(stringInterpolation
      :)(rax_13)
61     rdx_7[0x3] = type metadata for String
62     *(rdx_7) = rax_20
63     int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(
      rax_12)
64     Type_18 rax_23 = &default argument 1 of print(_:
      separator:terminator:>()
65     Type_19 rax_24 = &default argument 2 of print(_:
      separator:terminator:>()
66     &print(_:separator:terminator:)(rax_22, rax_23,
      rax_24)
67     int64_t rax_26
68     rax_26.0x0 = add_overflow(v, var_48)
69     if (rax_26.0x0 && 0x1 != 0x0) {
70         break
71     }
72     var_48 = v + var_48
73 }
74 trap(0x6)
75 }

```

A.0.4 RAIIOp. Similar to ReferenceCountingOp, removes calls to object destructors.

```

1 void f(d:)(void* arg1) {
2     Type_2 var_48
3     Type_3 var_70
4     Type_14 rax_30
5     Type_21 rax_13
6     Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1,
      type metadata for Any + 0x8)
7     Type_10 rax_1 = &String.init(_builtinStringLiteral:
      utf8CodeUnitCount:isASCII:)(
      "Entries:", 0x8, 0x1)
8     rdx[0x3] = type metadata for String
9     *(rdx) = rax_1
10    int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
      rax)
11    Type_11 rax_4 = &default argument 1 of print(_:
      separator:terminator:>()
12    Type_12 rax_5 = &default argument 2 of print(_:
      separator:terminator:>()

```

```

13     &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
14     var_48 = 0x0
15     void var_38
16     &Dictionary.makeIterator()(&var_38, arg1, type
      metadata for Int, type metadata for Int, protocol
      witness table for Int)
17     &memcpy(&var_70, &var_38, 0x28)
18     while (0x1) {
19         option_tup_t var_88
20         &Dictionary.Iterator.next()(&var_88, &var_70, &
      __swift_instantiateConcreteTypeFromMangledName(&
      demangling cache variable for type metadata for [Int
      : Int].Iterator))
21         int64_t k = var_88.0x0
22         int64_t v = var_88.0x8
23         int64_t rax_9
24         rax_9.0x0 = var_88.0x10
25         if (rax_9.0x0 && 0x1 != 0x0) {
26             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))
27             Type_13 rax_29 = &_allocateUninitializedArray
      <A>(_:)(0x1)
28             Type_14 rax_30 = &DefaultStringInterpolation.
      init(literalCapacity:interpolationCount:)(0x9, 0x1)
29             int32_t var_204 = 0x1
30             Type_15 rax_31 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
      "Average: ", 0x9, 0x1)
31             &DefaultStringInterpolation.appendLiteral(_:)(
      &rax_30, rax_31)
32             float var_a8 = zmm0_1
33             &DefaultStringInterpolation.
      appendInterpolation<A>(_:)(&rax_30, &var_a8, type
      metadata for Float, protocol witness table for Float
      , protocol witness table for Float)
34             Type_16 rax_32 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
      data_100003f7f, 0x0, 0x1)
35             &DefaultStringInterpolation.appendLiteral(_:)(
      &rax_30, rax_32)
36             Type_17 rax_34 = &String.init(
      stringInterpolation:)(rax_30)
37             rdx_20[0x3] = type metadata for String
38             *(rdx_20) = rax_34
39             int64_t rax_36 = &_finalizeUninitializedArray
      <A>(_:)(rax_29)
40             Type_18 rax_37 = &default argument 1 of print
      (_:separator:terminator:>()
41             Type_19 rax_38 = &default argument 2 of print
      (_:separator:terminator:>()
42             &print(_:separator:terminator:)(rax_36,
      rax_37, rax_38)
43             return
44         }
45         Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(
      0x1, type metadata for Any + 0x8)
46         Type_21 rax_13 = &DefaultStringInterpolation.init
      (literalCapacity:interpolationCount:)(0x4, 0x2)
47         Type_22 rax_14 = &String.init(
      _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
      ", 0x2, 0x1)
48         &DefaultStringInterpolation.appendLiteral(_:)(&
      rax_13, rax_14)
49         int64_t k_1 = k
50         &DefaultStringInterpolation.appendInterpolation<A
      >(_:)(&rax_13, &k_1, type metadata for Int, protocol
      witness table for Int)

```

```

51     Type_23 rax_16 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(":
        ", 0x2, 0x1)
52     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_16)
53     int64_t v_1 = v
54     &DefaultStringInterpolation.appendInterpolation<A
        >(_:)(&rax_13, &v_1, type metadata for Int, protocol
        witness table for Int)
55     Type_24 rax_18 = &String.init(
        _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
        data_100003f7f, 0x0, 0x1)
56     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_18)
57     Type_25 rax_20 = &String.init(stringInterpolation
        :)(rax_13)
58     rdx_7[0x3] = type metadata for String
59     *(rdx_7) = rax_20
60     int64_t rax_22 = &_amp;finalizeUninitializedArray<A>(_:)(
        rax_12)
61     Type_26 rax_23 = &default argument 1 of print(_:
        separator:terminator:>()
62     Type_27 rax_24 = &default argument 2 of print(_:
        separator:terminator:>()
63     &print(_:separator:terminator:)(rax_22, rax_23,
        rax_24)
64     int64_t rax_26
65     rax_26.0x0 = add_overflow(v, var_48)
66     if (rax_26.0x0 && 0x1 != 0x0) {
67         break
68     }
69     var_48 = v + var_48
70 }
71 trap(0x6)
72 }

```

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).

```

1 void f(d:)(void* arg1) {
2     Type_21 var_48
3     Type_22 var_70
4     Type_6 rax_30
5     Type_13 rax_13
6     Type_1 rax = &_amp;allocateUninitializedArray<A>(_:)(0x1,
        type metadata for Any + 0x8)
7     String rax_1 = "Entries:"
8     rdx[0x3] = type metadata for String
9     *(rdx) = rax_1
10    int64_t rax_3 = &_amp;finalizeUninitializedArray<A>(_:)(
        rax)
11    Type_3 rax_4 = &default argument 1 of print(_:
        separator:terminator:>()
12    Type_4 rax_5 = &default argument 2 of print(_:
        separator:terminator:>()
13    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
14    var_48 = 0x0
15    void var_38
16    &Dictionary.makeIterator()(&var_38, arg1, type
        metadata for Int, type metadata for Int, protocol
        witness table for Int)
17    &memcpy(&var_70, &var_38, 0x28)
18    while (0x1) {
19        option_tup_t var_88
20        &Dictionary.Iterator.next()(&var_88, &var_70, &
        ___swift_instantiateConcreteTypeFromMangledName(&
        demangling cache variable for type metadata for [Int
        : Int].Iterator))

```

```

21    int64_t k = var_88.0x0
22    int64_t v = var_88.0x8
23    int64_t rax_9
24    rax_9.0x0 = var_88.0x10
25    if (rax_9.0x0 && 0x1 != 0x0) {
26        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))
27        Type_5 rax_29 = &_amp;allocateUninitializedArray<
        A>(_:)(0x1)
28        Type_6 rax_30 = &DefaultStringInterpolation.
        init(literalCapacity:interpolationCount:)(0x9, 0x1)
29        int32_t var_204 = 0x1
30        String rax_31 = "Average: "
31        &DefaultStringInterpolation.appendLiteral(_:)(
        &rax_30, rax_31)
32        float var_a8 = zmm0_1
33        &DefaultStringInterpolation.
        appendInterpolation<A>(_:)(&rax_30, &var_a8, type
        metadata for Float, protocol witness table for Float
        , protocol witness table for Float)
34        String rax_32 = ""
35        &DefaultStringInterpolation.appendLiteral(_:)(
        &rax_30, rax_32)
36        Type_9 rax_34 = &String.init(
        stringInterpolation:)(rax_30)
37        rdx_20[0x3] = type metadata for String
38        *(rdx_20) = rax_34
39        int64_t rax_36 = &_amp;finalizeUninitializedArray
        <A>(_:)(rax_29)
40        Type_10 rax_37 = &default argument 1 of print
        (_:separator:terminator:>()
41        Type_11 rax_38 = &default argument 2 of print
        (_:separator:terminator:>()
42        &print(_:separator:terminator:)(rax_36,
        rax_37, rax_38)
43        return
44    }
45    Type_12 rax_12 = &_amp;allocateUninitializedArray<A>(_:)(
        0x1, type metadata for Any + 0x8)
46    Type_13 rax_13 = &DefaultStringInterpolation.init
        (literalCapacity:interpolationCount:)(0x4, 0x2)
47    String rax_14 = " "
48    &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_14)
49    int64_t k_1 = k
50    &DefaultStringInterpolation.appendInterpolation<A
        >(_:)(&rax_13, &k_1, type metadata for Int, protocol
        witness table for Int)
51    String rax_16 = ": "
52    &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_16)
53    int64_t v_1 = v
54    &DefaultStringInterpolation.appendInterpolation<A
        >(_:)(&rax_13, &v_1, type metadata for Int, protocol
        witness table for Int)
55    String rax_18 = ""
56    &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, rax_18)
57    Type_17 rax_20 = &String.init(stringInterpolation
        :)(rax_13)
58    rdx_7[0x3] = type metadata for String
59    *(rdx_7) = rax_20
60    int64_t rax_22 = &_amp;finalizeUninitializedArray<A>(_:)(
        rax_12)
61    Type_18 rax_23 = &default argument 1 of print(_:
        separator:terminator:>()

```



```

62     Type_19 rax_24 = &default argument 2 of print(_:
        separator:terminator:>()
63     &print(_:separator:terminator:)(rax_22, rax_23,
        rax_24)
64     int64_t rax_26
65     rax_26.0x0 = add_overflow(v, var_48)
66     if (rax_26.0x0 && 0x1 != 0x0) {
67         break
68     }
69     var_48 = v + var_48
70 }
71 trap(0x6)
72 }

```

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

```

1 void f(d:)(void* arg1) {
2     Type_2 var_48
3     Type_3 var_70
4     Type_14 rax_30
5     Type_21 rax_13
6     Type_9 rax = &_allocateUninitializedArray<A>(_:)(0x1,
        type metadata for Any + 0x8)
7     rdx[0x3] = type metadata for String
8     *(rdx) = "Entries:"
9     int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
        rax)
10    Type_11 rax_4 = &default argument 1 of print(_:
        separator:terminator:>()
11    Type_12 rax_5 = &default argument 2 of print(_:
        separator:terminator:>()
12    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
13    var_48 = 0x0
14    void var_38
15    &Dictionary.makeIterator()(&var_38, arg1, type
        metadata for Int, type metadata for Int, protocol
        witness table for Int)
16    &memcpy(&var_70, &var_38, 0x28)
17    while (0x1) {
18        option_tup_t var_88
19        &Dictionary.Iterator.next()(&var_88, &var_70, &
        __swift_instantiateConcreteTypeFromMangledName(&
        demangling cache variable for type metadata for [Int
        : Int].Iterator))
20        int64_t k = var_88.0x0
21        int64_t v = var_88.0x8
22        int64_t rax_9
23        rax_9.0x0 = var_88.0x10
24        if (rax_9.0x0 && 0x1 != 0x0) {
25            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))
26            Type_13 rax_29 = &_allocateUninitializedArray
        <A>(_:)(0x1)
27            Type_14 rax_30 = &DefaultStringInterpolation.
        init(literalCapacity:interpolationCount:)(0x9, 0x1)
28            int32_t var_204 = 0x1
29            &DefaultStringInterpolation.appendLiteral(_:)(
        &rax_30, "Average: ")
30            float var_a8 = zmm0_1
31            &DefaultStringInterpolation.
        appendInterpolation<A>(_:)(&rax_30, &var_a8, type
        metadata for Float, protocol witness table for Float
        , protocol witness table for Float)
32            &DefaultStringInterpolation.appendLiteral(_:)(
        &rax_30, "")
33            Type_17 rax_34 = &String.init(
        stringInterpolation:)(rax_30)

```

```

34     rdx_20[0x3] = type metadata for String
35     *(rdx_20) = rax_34
36     int64_t rax_36 = &_finalizeUninitializedArray
        <A>(_:)(rax_29)
37     Type_18 rax_37 = &default argument 1 of print
        (_:separator:terminator:>()
38     Type_19 rax_38 = &default argument 2 of print
        (_:separator:terminator:>()
39     &print(_:separator:terminator:)(rax_36,
        rax_37, rax_38)
40     return
41 }
42     Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(
        0x1, type metadata for Any + 0x8)
43     Type_21 rax_13 = &DefaultStringInterpolation.init
        (literalCapacity:interpolationCount:)(0x4, 0x2)
44     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, " ")
45     int64_t k_1 = k
46     &DefaultStringInterpolation.appendInterpolation<A>
        >(_:)(&rax_13, &k_1, type metadata for Int, protocol
        witness table for Int)
47     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, ": ")
48     int64_t v_1 = v
49     &DefaultStringInterpolation.appendInterpolation<A>
        >(_:)(&rax_13, &v_1, type metadata for Int, protocol
        witness table for Int)
50     &DefaultStringInterpolation.appendLiteral(_:)(&
        rax_13, "")
51     Type_25 rax_20 = &String.init(stringInterpolation
        :)(rax_13)
52     rdx_7[0x3] = type metadata for String
53     *(rdx_7) = rax_20
54     int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(
        rax_12)
55     Type_26 rax_23 = &default argument 1 of print(_:
        separator:terminator:>()
56     Type_27 rax_24 = &default argument 2 of print(_:
        separator:terminator:>()
57     &print(_:separator:terminator:)(rax_22, rax_23,
        rax_24)
58     int64_t rax_26
59     rax_26.0x0 = add_overflow(v, var_48)
60     if (rax_26.0x0 && 0x1 != 0x0) {
61         break
62     }
63     var_48 = v + var_48
64 }
65 trap(0x6)
66 }

```

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

```

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

```

```

11     Type_12 rax_5 = &default argument 2 of print(_:
    separator:terminator:>()
12     &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
13     var_48 = 0x0
14     void var_38
15     &Dictionary.makeIterator()(&var_38, arg1, type
    metadata for Int, type metadata for Int, protocol
    witness table for Int)
16     &memcpy(&var_70, &var_38, 0x28)
17     while (0x1) {
18         option_tup_t var_88
19         &Dictionary.Iterator.next()(&var_88, &var_70, &
    __swift_instantiateConcreteTypeFromMangledName(&
    demangling cache variable for type metadata for [Int
    : Int].Iterator))
20         int64_t k = var_88.0x0
21         int64_t v = var_88.0x8
22         int64_t rax_9
23         rax_9.0x0 = var_88.0x10
24         if (rax_9.0x0 && 0x1 != 0x0) {
25             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))
26             Type_13 rax_29 = &_allocateUninitializedArray
    <A>(_:)(0x1)
27             Type_14 rax_30 = &DefaultStringInterpolation.
    init(literalCapacity:interpolationCount:)(0x9, 0x1)
28             int32_t var_204 = 0x1
29             &DefaultStringInterpolation.appendLiteral(_:)(
    &rax_30, "Average: ")
30             float var_a8 = zmm0_1
31             &DefaultStringInterpolation.
    appendInterpolation<A>(_:)(&rax_30, &var_a8, type
    metadata for Float, protocol witness table for Float
    , protocol witness table for Float)
32             &DefaultStringInterpolation.appendLiteral(_:)(
    &rax_30, "")
33             String rax_34 = String(rax_30)
34             rdx_20[0x3] = type metadata for String
35             *(rdx_20) = rax_34
36             int64_t rax_36 = &_finalizeUninitializedArray
    <A>(_:)(rax_29)
37             Type_18 rax_37 = &default argument 1 of print
    (_:separator:terminator:>()
38             Type_19 rax_38 = &default argument 2 of print
    (_:separator:terminator:>()
39             &print(_:separator:terminator:)(rax_36,
    rax_37, rax_38)
40             return
41         }
42         Type_20 rax_12 = &_allocateUninitializedArray<A>(
    _:)(0x1, type metadata for Any + 0x8)
43         Type_21 rax_13 = &DefaultStringInterpolation.init
    (literalCapacity:interpolationCount:)(0x4, 0x2)
44         &DefaultStringInterpolation.appendLiteral(_:)(&
    rax_13, " ")
45         int64_t k_1 = k
46         &DefaultStringInterpolation.appendInterpolation<A
    >(_:)(&rax_13, &k_1, type metadata for Int, protocol
    witness table for Int)
47         &DefaultStringInterpolation.appendLiteral(_:)(&
    rax_13, ": ")
48         int64_t v_1 = v
49         &DefaultStringInterpolation.appendInterpolation<A
    >(_:)(&rax_13, &v_1, type metadata for Int, protocol
    witness table for Int)
50         &DefaultStringInterpolation.appendLiteral(_:)(&
    rax_13, "")

```

```

51     String rax_20 = String(rax_13)
52     rdx_7[0x3] = type metadata for String
53     *(rdx_7) = rax_20
54     int64_t rax_22 = &_finalizeUninitializedArray<A>(
    _:)(rax_12)
55     Type_26 rax_23 = &default argument 1 of print(_:
    separator:terminator:>()
56     Type_27 rax_24 = &default argument 2 of print(_:
    separator:terminator:>()
57     &print(_:separator:terminator:)(rax_22, rax_23,
    rax_24)
58     int64_t rax_26
59     rax_26.0x0 = add_overflow(v, var_48)
60     if (rax_26.0x0 && 0x1 != 0x0) {
61         break
62     }
63     var_48 = v + var_48
64 }
65 trap(0x6)
66 }

```

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

```

1 void f(d:)(void* arg1) {
2     Type_21 var_48
3     Type_22 var_70
4     Type_6 rax_30
5     Type_13 rax_13
6     Type_1 rax = &_allocateUninitializedArray<A>(_:)(0x1,
    type metadata for Any + 0x8)
7     rdx[0x3] = type metadata for String
8     *(rdx) = "Entries:"
9     int64_t rax_3 = &_finalizeUninitializedArray<A>(_:)(
    rax)
10    Type_3 rax_4 = &default argument 1 of print(_:
    separator:terminator:>()
11    Type_4 rax_5 = &default argument 2 of print(_:
    separator:terminator:>()
12    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
13    var_48 = 0x0
14    void var_38
15    &Dictionary.makeIterator()(&var_38, arg1, type
    metadata for Int, type metadata for Int, protocol
    witness table for Int)
16    &memcpy(&var_70, &var_38, 0x28)
17    while (0x1) {
18        option_tup_t var_88
19        &Dictionary.Iterator.next()(&var_88, &var_70, &
    __swift_instantiateConcreteTypeFromMangledName(&
    demangling cache variable for type metadata for [Int
    : Int].Iterator))
20        int64_t k = var_88.0x0
21        int64_t v = var_88.0x8
22        int64_t rax_9
23        rax_9.0x0 = var_88.0x10
24        if (rax_9.0x0 && 0x1 != 0x0) {
25            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))
26            Type_5 rax_29 = &_allocateUninitializedArray<
    A>(_:)(0x1)
27            Type_6 rax_30 = &DefaultStringInterpolation.
    init(literalCapacity:interpolationCount:)(0x9, 0x1)
28            int32_t var_204 = 0x1
29            rax_30.append("Average: ")
30            float var_a8 = zmm0_1

```

```

31     rax_30.append(var_a8)
32     rax_30.append("")
33     String rax_34 = String(rax_30)
34     rdx_20[0x3] = type metadata for String
35     *(rdx_20) = rax_34
36     int64_t rax_36 = &_finalizeUninitializedArray
37     <A>(_:)(rax_29)
38     Type_10 rax_37 = &default argument 1 of print
39     (_:separator:terminator:>()
40     Type_11 rax_38 = &default argument 2 of print
41     (_:separator:terminator:>()
42     &print(_:separator:terminator:)(rax_36,
43     rax_37, rax_38)
44     return
45 }
46 Type_12 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
47 Type_13 rax_13 = &DefaultStringInterpolation.init
48 (literalCapacity:interpolationCount:)(0x4, 0x2)
49 rax_13.append(" ")
50 int64_t k_1 = k
51 rax_13.append(k_1)
52 rax_13.append(": ")
53 int64_t v_1 = v
54 rax_13.append(v_1)
55 rax_13.append("")
56 String rax_20 = String(rax_13)
57 rdx_7[0x3] = type metadata for String
58 *(rdx_7) = rax_20
59 int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
60 Type_18 rax_23 = &default argument 1 of print(_:
61 separator:terminator:>()
62 Type_19 rax_24 = &default argument 2 of print(_:
63 separator:terminator:>()
64 &print(_:separator:terminator:)(rax_22, rax_23,
65 rax_24)
66 int64_t rax_26
67 rax_26.0x0 = add_overflow(v, var_48)
68 if (rax_26.0x0 && 0x1 != 0x0) {
69     break
70 }
71 var_48 = v + var_48
72 }
73 trap(0x6)
74 }

```

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.

```

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

```

```

11    Type_12 rax_5 = &default argument 2 of print(_:
12    separator:terminator:>()
13    &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
14    var_48 = 0x0
15    void var_38
16    &Dictionary.makeIterator()(&var_38, arg1, type
17    metadata for Int, type metadata for Int, protocol
18    witness table for Int)
19    &memcpy(&var_70, &var_38, 0x28)
20    while (0x1) {
21        option_tup_t var_88
22        &Dictionary.Iterator.next()(&var_88, &var_70, &
23        __swift_instantiateConcreteTypeFromMangledName(&
24        demangling cache variable for type metadata for [Int
25        : Int].Iterator))
26        int64_t k = var_88.0x0
27        int64_t v = var_88.0x8
28        int64_t rax_9
29        rax_9.0x0 = var_88.0x10
30        if (rax_9.0x0 && 0x1 != 0x0) {
31            float zmm0_1 = float.s(var_48) f/ float.s(&
32            Dictionary.count.getter(arg1, type metadata for Int,
33            type metadata for Int, protocol witness table for
34            Int))
35            float var_a8 = zmm0_1
36            int32_t var_204 = 0x1
37            Type_13 rax_29 = &_allocateUninitializedArray
38            <A>(_:)(0x1)
39            String rax_34 = "Average: \"(var_a8)\"
40            rdx_20[0x3] = type metadata for String
41            *(rdx_20) = rax_34
42            int64_t rax_36 = &_finalizeUninitializedArray
43            <A>(_:)(rax_29)
44            Type_18 rax_37 = &default argument 1 of print
45            (_:separator:terminator:>()
46            Type_19 rax_38 = &default argument 2 of print
47            (_:separator:terminator:>()
48            &print(_:separator:terminator:)(rax_36,
49            rax_37, rax_38)
50            return
51        }
52        int64_t v_1 = v
53        int64_t k_1 = k
54        Type_20 rax_12 = &_allocateUninitializedArray<A>(_:)(0x1, type metadata for Any + 0x8)
55        String rax_20 = " \"(k_1): \"(v_1)\"
56        rdx_7[0x3] = type metadata for String
57        *(rdx_7) = rax_20
58        int64_t rax_22 = &_finalizeUninitializedArray<A>(_:)(rax_12)
59        Type_26 rax_23 = &default argument 1 of print(_:
60        separator:terminator:>()
61        Type_27 rax_24 = &default argument 2 of print(_:
62        separator:terminator:>()
63        &print(_:separator:terminator:)(rax_22, rax_23,
64        rax_24)
65        int64_t rax_26
66        rax_26.0x0 = add_overflow(v, var_48)
67        if (rax_26.0x0 && 0x1 != 0x0) {
68            break
69        }
70        var_48 = v + var_48
71    }
72    trap(0x6)
73 }

```

A.0.10 InterpConstructProp. Propagates interpolated strings.

```

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

```

```

2   Type_21 var_48
3   Type_22 var_70
4   Type_1 rax = &allocateUninitializedArray<A>(_:)(0x1,
   type metadata for Any + 0x8)
5   rdx[0x3] = type metadata for String
6   *(rdx) = "Entries:"
7   int64_t rax_3 = &finalizeUninitializedArray<A>(_:)(
   rax)
8   Type_3 rax_4 = &default argument 1 of print(_:
   separator:terminator:>()
9   Type_4 rax_5 = &default argument 2 of print(_:
   separator:terminator:>()
10  &print(_:separator:terminator:)(rax_3, rax_4, rax_5)
11  var_48 = 0x0
12  void var_38
13  &Dictionary.makeIterator()(&var_38, arg1, type
   metadata for Int, type metadata for Int, protocol
   witness table for Int)
14  &memcpy(&var_70, &var_38, 0x28)
15  while (0x1) {
16      option_tup_t var_88
17      &Dictionary.Iterator.next()(&var_88, &var_70, &
   ___swift_instantiateConcreteTypeFromMangledName(&
   demangling cache variable for type metadata for [Int
   : Int].Iterator))
18      int64_t k = var_88.0x0
19      int64_t v = var_88.0x8
20      int64_t rax_9
21      rax_9.0x0 = var_88.0x10
22      if (rax_9.0x0 && 0x1 != 0x0) {
23          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))
24          float var_a8 = zmm0_1
25          int32_t var_204 = 0x1
26          Type_5 rax_29 = &allocateUninitializedArray<
   A>(_:)(0x1)
27          rdx_20[0x3] = type metadata for String
28          *(rdx_20) = "Average: \"(var_a8)\"
29          int64_t rax_36 = &finalizeUninitializedArray
   <A>(_:)(rax_29)
30          Type_10 rax_37 = &default argument 1 of print
   (_:separator:terminator:>()
31          Type_11 rax_38 = &default argument 2 of print
   (_:separator:terminator:>()
32          &print(_:separator:terminator:)(rax_36,
   rax_37, rax_38)
33          return
34      }
35      int64_t v_1 = v
36      int64_t k_1 = k
37      Type_12 rax_12 = &allocateUninitializedArray<A>(_:
   _:)(0x1, type metadata for Any + 0x8)
38      rdx_7[0x3] = type metadata for String
39      *(rdx_7) = " \"(k_1): \"(v_1)\"
40      int64_t rax_22 = &finalizeUninitializedArray<A>(_:
   _:)(rax_12)
41      Type_18 rax_23 = &default argument 1 of print(_:
   separator:terminator:>()
42      Type_19 rax_24 = &default argument 2 of print(_:
   separator:terminator:>()
43      &print(_:separator:terminator:)(rax_22, rax_23,
   rax_24)
44      int64_t rax_26
45      rax_26.0x0 = add_overflow(v, var_48)
46      if (rax_26.0x0 && 0x1 != 0x0) {
47          break
48      }

```

```

49     var_48 = v + var_48
50 }
51     trap(0x6)
52 }

```

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

```

1 void f(d:)(void* arg1) {
2     Type_2 var_48
3     Type_3 var_70
4     print("Entries:")
5     var_48 = 0x0
6     void var_38
7     &Dictionary.makeIterator()(&var_38, arg1, type
   metadata for Int, type metadata for Int, protocol
   witness table for Int)
8     &memcpy(&var_70, &var_38, 0x28)
9     while (0x1) {
10         option_tup_t var_88
11         &Dictionary.Iterator.next()(&var_88, &var_70, &
   ___swift_instantiateConcreteTypeFromMangledName(&
   demangling cache variable for type metadata for [Int
   : Int].Iterator))
12         int64_t k = var_88.0x0
13         int64_t v = var_88.0x8
14         int64_t rax_9
15         rax_9.0x0 = var_88.0x10
16         if (rax_9.0x0 && 0x1 != 0x0) {
17             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))
18             float var_a8 = zmm0_1
19             int32_t var_204 = 0x1
20             print("Average: \"(var_a8)\"
21             return
22         }
23         int64_t v_1 = v
24         int64_t k_1 = k
25         print(" \"(k_1): \"(v_1)\"
26         int64_t rax_26
27         rax_26.0x0 = add_overflow(v, var_48)
28         if (rax_26.0x0 && 0x1 != 0x0) {
29             break
30         }
31         var_48 = v + var_48
32     }
33     trap(0x6)
34 }

```

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

```

1 void f(d:)(void* arg1) {
2     Type_2 var_48
3     Type_3 var_70
4     print("Entries:")
5     var_48 = 0x0
6     void var_38
7     &Dictionary.makeIterator()(&var_38, arg1, type
   metadata for Int, type metadata for Int, protocol
   witness table for Int)
8     &memcpy(&var_70, &var_38, 0x28)
9     loop {
10         option_tup_t var_88

```

```

11      &Dictionary.Iterator.next()(&var_88, &var_70, &
    __swift_instantiateConcreteTypeFromMangledName(&
    demangling_cache_variable_for_type_metadata_for[Int
    : Int].Iterator))
12      int64_t k = var_88.0x0
13      int64_t v = var_88.0x8
14      int64_t rax_9
15      rax_9.0x0 = var_88.0x10
16      if (rax_9.0x0 && 0x1 != 0x0) {
17          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))
18          float var_a8 = zmm0_1
19          int32_t var_204 = 0x1
20          print("Average: \(\var_a8)")
21          return
22      }
23      int64_t v_1 = v
24      int64_t k_1 = k
25      print(" \(\k_1): \(\v_1)")
26      int64_t rax_26
27      rax_26.0x0 = add_overflow(v, var_48)
28      if (rax_26.0x0 && 0x1 != 0x0) {
29          trap(0x6)
30      }
31      var_48 = v + var_48
32  }
33 }

```

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.

```

1 void f(d:)(void* arg1) {
2     int64_t var_48
3     Type_3 var_70
4     print("Entries:")
5     var_48 = 0x0
6     void var_38
7     &Dictionary.makeIterator()(&var_38, arg1, type
    metadata_for Int, type_metadata_for Int, protocol
    witness_table_for Int)
8     &memcpy(&var_70, &var_38, 0x28)
9     loop {
10         option_tup_t var_88
11         &Dictionary.Iterator.next()(&var_88, &var_70, &
    __swift_instantiateConcreteTypeFromMangledName(&
    demangling_cache_variable_for_type_metadata_for[Int
    : Int].Iterator))
12         int64_t k = var_88.0x0
13         int64_t v = var_88.0x8
14         int64_t rax_9
15         rax_9.0x0 = var_88.0x10
16         if (rax_9.0x0 && 0x1 != 0x0) {
17             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))
18             float var_a8 = zmm0_1
19             int32_t var_204 = 0x1
20             print("Average: \(\var_a8)")
21             return
22         }
23         int64_t v_1 = v
24         int64_t k_1 = k
25         print(" \(\k_1): \(\v_1)")
26         var_48 = v + var_48
27     }

```

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.

```

1 void f(d:)(void* arg1) {
2     int64_t var_48
3     Type_22 var_70
4     print("Entries:")
5     var_48 = 0x0
6     void var_38
7     &Dictionary.makeIterator()(&var_38, arg1, type
    metadata_for Int, type_metadata_for Int, protocol
    witness_table_for Int)
8     &memcpy(&var_70, &var_38, 0x28)
9     loop {
10         option_tup_t var_88
11         &Dictionary.Iterator.next()(&var_88, &var_70, &
    __swift_instantiateConcreteTypeFromMangledName(&
    demangling_cache_variable_for_type_metadata_for[Int
    : Int].Iterator))
12         int64_t k = var_88.0x0
13         int64_t v = var_88.0x8
14         int64_t rax_9
15         rax_9.0x0 = var_88.0x10
16         if (rax_9.0x0 && 0x1 != 0x0) {
17             break
18         }
19         int64_t v_1 = v
20         int64_t k_1 = k
21         print(" \(\k_1): \(\v_1)")
22         var_48 = v + var_48
23     }
24     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))
25     float var_a8 = zmm0_1
26     int32_t var_204 = 0x1
27     print("Average: \(\var_a8)")
28     return
29 }

```

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 `IteratorConstruct` followed by an infinite loop where the first statement is an `IteratorNext` followed by an `OptionalUnwrap`.

After the original rule is applied:

```

1 void f(d:)([Int: Int] arg1) {
2     int64_t var_48
3     Iterator<(Int, Int)> var_38
4     print("Entries:")
5     var_48 = 0x0
6     Iterator<(Int, Int)> var_38 = arg1.makeIterator()

```



```

7      loop {
8          option_tup_t var_88
9          &Dictionary.Iterator.next()(&var_88, &var_38, &
10         __swift_instantiateConcreteTypeFromMangledName(&
11         demangling cache variable for type metadata for [Int
12         : Int].Iterator))
13         int64_t k = var_88.0x0
14         int64_t v = var_88.0x8
15         int64_t rax_9
16         rax_9.0x0 = var_88.0x10
17         if (rax_9.0x0 && 0x1 != 0x0) {
18             break
19         }
20         int64_t v_1 = v
21         int64_t k_1 = k
22         print(" \k_1): \v_1)")
23         var_48 = v + var_48
24     }
25     float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
26     .count.getter(arg1, type metadata for Int, type
27     metadata for Int, protocol witness table for Int))
28     float var_a8 = zmm0_1
29     int32_t var_204 = 0x1
30     print("Average: \var_a8)")
31     return
32 }

```

After IteratorNext is applied:

```

1 void f(d:)([Int: Int] arg1) {
2     int64_t var_48
3     Iterator<(Int, Int)> var_38
4     print("Entries:")
5     var_48 = 0x0
6     Iterator<(Int, Int)> var_38 = arg1.makeIterator()
7     loop {
8         (Int, Int)? var_88
9         var_88 = var_38.next()
10        int64_t k = var_88.0x0
11        int64_t v = var_88.0x8
12        int64_t rax_9
13        rax_9.0x0 = var_88.0x10
14        if (rax_9.0x0 && 0x1 != 0x0) {
15            break
16        }
17        int64_t v_1 = v
18        int64_t k_1 = k
19        print(" \k_1): \v_1)")
20        var_48 = v + var_48
21    }
22    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
23    .count.getter(arg1, type metadata for Int, type
24    metadata for Int, protocol witness table for Int))
25    float var_a8 = zmm0_1
26    int32_t var_204 = 0x1
27    print("Average: \var_a8)")
28    return
29 }

```

After OptionalUnwrap is applied:

```

1 void f(d:)([Int: Int] arg1) {
2     Int var_48
3     print("Entries:")
4     var_48 = 0x0
5     Iterator<(Int, Int)> var_38 = arg1.makeIterator()
6     loop {
7         (Int, Int)? var_88 = var_38.next()
8         (Int, Int) (k, v) = var_88
9         if (var_88 == nil) {
10             break

```

```

11        }
12        print(" \k): \v)")
13        var_48 = v + var_48
14    }
15    float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
16    .count.getter(arg1, type metadata for Int, type
17    metadata for Int, protocol witness table for Int))
18    float var_a8 = zmm0_1
19    int32_t var_204 = 0x1
20    print("Average: \var_a8)")
21    return
22 }

```

After IteratorLoop is applied:

```

1 void f(d:)([Int: Int] arg1) {
2     Int var_48
3     print("Entries:")
4     var_48 = 0x0
5     for (k, v) in arg1 {
6         print(" \k): \v)")
7         var_48 = v + var_48
8     }
9     float zmm0_1 = float.s(var_48) f/ float.s(&Dictionary
10    .count.getter(arg1, type metadata for Int, type
11    metadata for Int, protocol witness table for Int))
12    float var_a8 = zmm0_1
13    int32_t var_204 = 0x1
14    print("Average: \var_a8)")
15    return
16 }

```

A.0.16 DictCount. Detects calls to Dictionary.count.getter.

```

1 void f(d:)([Int: Int] arg1) {
2     Int var_48
3     print("Entries:")
4     var_48 = 0x0
5     for (k, v) in arg1 {
6         print(" \k): \v)")
7         var_48 = v + var_48
8     }
9     float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
10    float var_a8 = zmm0_1
11    int32_t var_204 = 0x1
12    print("Average: \var_a8)")
13    return
14 }

```

A.0.17 SwiftVarDecl. Overrides the h11.VarDecl and h11.VarInit descriptors to display these statements (when assigning a variable with a Swift type) with Swift syntax.

```

1 void f(d:)([Int: Int] arg1) {
2     var var_48: Int
3     print("Entries:")
4     var_48 = 0x0
5     for (k, v) in arg1 {
6         print(" \k): \v)")
7         var_48 = v + var_48
8     }
9     float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
10    float var_a8 = zmm0_1
11    int32_t var_204 = 0x1
12    print("Average: \var_a8)")
13    return
14 }

```

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

```

1 func f(arg1: [Int: Int]) -> void {
2     var var_48: Int
3     print("Entries:")
4     var_48 = 0x0
5     for (k, v) in arg1 {
6         print("  \<k>: \<v>")
7         var_48 = v + var_48
8     }
9     float zmm0_1 = float.s(var_48) f/ float.s(arg1.count)
10    float var_a8 = zmm0_1
11    int32_t var_204 = 0x1
12    print("Average: \<var_a8>")
13    return
14 }
```

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
 - * DoWhile
 - * 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.

```

1 import Foundation
2 import CryptoKit
3
4 struct Input {
5     static func prompt(_ message: String) -> String {
6         print(message, terminator: ": ")
7         return readLine() ?? ""
8     }
9 }
10
11 enum ValidationResult {
12     case success(String)
13     case failure(String)
14 }
15
16 struct KeyValidator {
17     private let expectedHash: String
18
19     init(expectedHash: String) {
20         self.expectedHash = expectedHash
21     }
22
23     func validate(_ input: String) -> ValidationResult {
24         var inputRev = ""
25
26         for c in input {
27             inputRev = "\<c>\<inputRev>"
28         }
29
30         let hash = SHA256.hash(data: Data(inputRev.utf8))
31         .compactMap {
32             String(format: "%02x", $0)
33         }.joined()
34
35         if hash == expectedHash {
36             return .success("Access Granted")
37         } else {
38             return .failure("Invalid key")
39         }
40     }
41 }
42
43 struct CrackMe {
44     private let validator: KeyValidator
45
46     init() {
47         let expectedHash = SHA256.hash(data: Data("foobar"
48             ".utf8)).compactMap {
49                 String(format: "%02x", $0)
50             }.joined()
51         self.validator = KeyValidator(expectedHash:
52             expectedHash)
```

```

50     }
51
52     func run() -> Bool {
53         print("Welcome to the Swift CrackMe!")
54         let key = Input.prompt("Enter the secret key")
55
56         switch validator.validate(key) {
57             case .success(let message):
58                 print(message)
59                 return true
60             case .failure(let message):
61                 print(message)
62                 return false
63         }
64     }
65 }
66
67 func main() {
68     let challenge = CrackMe()
69     if !challenge.run() {
70         exit(EXIT_FAILURE)
71     }
72 }
73
74 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

```

1  int64_t KeyValidator.validate(_:)(int64_t arg1, int64_t
   arg2, int64_t arg3, int64_t arg4)
2      int512_t zmm0
3      zmm0.o = zx.o(0)
4      int128_t var_148 = zx.o(0)
5      int256_t s_1
6      (&__builtin_memset)(s: &s_1, c: 0, n: 0x50)
7      int64_t var_f8 = 0
8      int128_t s
9      (&__builtin_memset)(s: &s, c: 0, n: 0x30)
10     void* rax = type metadata accessor for SHA256Digest
       (0, zmm0)
11     void* rax_1 = *(rax - 8)
12     int64_t rax_2 = *(rax_1 + 0x40)
13     int64_t rdx_1
14     int64_t rsi_1
15     int64_t rdi_1
16     rdx_1, rsi_1, rdi_1 = (&___chkstk_darwin)()
17     int64_t var_258
18     void* rsp = &var_258 - ((rax_2 + 0xf) & 0
       xffffffffffffffff)
19     int128_t var_28
20     var_28.q = rdi_1
21     var_28:8.q = rsi_1
22     int128_t var_38
23     var_38.q = rdx_1
24     var_38:8.q = arg4
25     int64_t rax_6
26     int64_t rdx_2

```

```

27     rax_6, rdx_2 = String.init(_builtinStringLiteral:
       utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0, 1)
28     int128_t var_48
29     var_48.q = rax_6
30     var_48:8.q = rdx_2
31     _swift_bridgeObjectRetain(arg2)
32     int64_t rax_7
33     int64_t rcx_1
34     int64_t rdx_3
35     int64_t r8_1
36     rax_7, rcx_1, rdx_3, r8_1 = String.makeIterator()(&
       arg1, arg2)
37     s_1.q = rax_7
38     s_1:8.q = rdx_3
39     s_1:0x10.q = rcx_1
40     s_1:0x18.q = r8_1
41     while (true)
42     {
43         int64_t rax_10
44         int64_t rdx_4
45         rax_10, rdx_4 = String.Iterator.next()(&s_1)
46         if (rdx_4 == 0)
47         {
48             break
49         }
50         int128_t var_b8_1
51         var_b8_1.q = rax_10
52         var_b8_1:8.q = rdx_4
53         int64_t rax_13
54         int64_t rdx_6
55         rax_13, rdx_6 = DefaultStringInterpolati...(
       literalCapacity:interpolationCount:)(0, 2)
56         s.q = rax_13
57         s:8.q = rdx_6
58         int64_t rax_14
59         int64_t rdx_7
60         rax_14, rdx_7 = String.init(_builtinStringLiteral
       :utf8CodeUnitCount:isASCII:)(" ", 0, 1)
61         DefaultStringInterpolation.appendLiteral(_:)(
       rax_14, rdx_7)
62         _swift_bridgeObjectRelease(rdx_7)
63         int64_t var_d8 = rax_10
64         int64_t var_d0_1 = rdx_4
65         DefaultStringInterpolation.appendInterpolation<A
       >(_:)(&var_d8, type metadata for Character, ...)
66         int64_t rax_16
67         int64_t rdx_9
68         rax_16, rdx_9 = String.init(_builtinStringLiteral
       :utf8CodeUnitCount:isASCII:)(" ", 0, 1)
69         DefaultStringInterpolation.appendLiteral(_:)(
       rax_16, rdx_9)
70         _swift_bridgeObjectRelease(rdx_9)
71         int64_t rax_17 = var_48.q
72         int64_t rdi_9 = var_48:8.q
73         _swift_bridgeObjectRetain(rdi_9)
74         int64_t var_e8 = rax_17
75         int64_t var_e0_1 = rdi_9
76         DefaultStringInterpolation.appendInterpolation<A
       >(_:)(&var_e8, type metadata for String, ...)
77         (&outlined_destroy_of_String)(&var_e8)
78         int64_t rax_19
79         int64_t rdx_11
80         rax_19, rdx_11 = String.init(
       _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
       " ", 0, 1)
81         DefaultStringInterpolation.appendLiteral(_:)(
       rax_19, rdx_11)
82         _swift_bridgeObjectRelease(rdx_11)
83         int64_t rax_20 = s.q

```

```

84     int64_t rdi_14 = s:8.q
85     _swift_bridgeObjectRetain(rdi_14)
86     (&outlined destroy of DefaultStringInterpolation)
87     (&s)
88     int64_t rax_21
89     int64_t rdx_12
90     rax_21, rdx_12 = String.init(stringInterpolation
91     :)(rax_20, rdi_14)
92     var_48.q = rax_21
93     var_48:8.q = rdx_12
94     _swift_bridgeObjectRelease(var_48:8.q)
95     _swift_bridgeObjectRelease(rdx_4)
96 }
97 (&outlined destroy of String.Iterator)(&s_1)
98 int64_t rax_23 = type metadata accessor for SHA256(0)
99 int64_t rax_24
100 int64_t rdx_13
101 rax_24, rdx_13 = String.utf8.getter(var_48.q, var_48
102 :8.q)
103 int64_t var_78 = rax_24
104 int64_t var_70 = rdx_13
105 int64_t rax_26
106 int64_t rdx_15
107 rax_26, rdx_15 = Data.init<A>(_:)(&var_78, type
108 metadata for String.UTF8View, ...)
109 int64_t var_88 = rax_26
110 int64_t var_80 = rdx_15
111 int64_t* var_1e0 = &var_88
112 static HashFunction.hash<A>(data:)(&var_88, rax_23,
113 type metadata for Data, ...)
114 (&outlined destroy of Data)(var_1e0)
115 (&___chkstk_darwin)()
116 *(rsp - 0x10) = &closure #1 in KeyValidator.validate(
117 _:)
118 *(rsp - 8) = 0
119 int64_t rax_33 = Sequence.compactMap<A>(_:)(&partial
120 apply for thunk ..., rsp - 0x20, rax, type metadata
121 for String, ...)
122 if (var_f8 != 0)
123 {
124     trap(6)
125 }
126 *(rax_1 + 8)(rsp, rax, rax_33, rsp)
127 int64_t var_98 = rax_33
128 int64_t rax_36 = (&
129 ___swift_instantiateConcreteTypeFromMangledName)(&
130 type metadata for [String])
131 int64_t rax_37 = (&lazy protocol witness ta...for
132 type [String] and conformance [A])()
133 int64_t rax_38
134 int64_t rdx_19
135 rax_38, rdx_19 = (&default argument 0 of Bi...
136 tionalCollection<>.joined(separator:))(rax_36)
137 int64_t rax_39
138 int64_t rdx_21
139 rax_39, rdx_21 = BidirectionalCollection<>.joined(
140 separator:)(rax_38, rdx_19, rax_36, rax_37)
141 _swift_bridgeObjectRelease(rdx_19)
142 (&outlined destroy of [String])(&var_98)
143 int128_t var_a8
144 var_a8.q = rax_39
145 var_a8:8.q = rdx_21
146 _swift_bridgeObjectRetain(arg4)
147 char rax_41 = static String.== infix(_:_:)(rax_39,
148 rdx_21, arg3, arg4)
149 _swift_bridgeObjectRelease(arg4)
150 int64_t var_238
151 char var_221
152 int64_t var_230

```

```

139     if ((rax_41 & 1) != 0)
140     {
141         int64_t rax_43
142         int64_t rdx_24
143         rax_43, rdx_24 = String.init(
144         _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
145         Access Granted ", 0x12, 0)
146         _swift_bridgeObjectRelease(rdx_21)
147         (&outlined destroy of String)(&var_48)
148         var_238 = rax_43
149         var_230 = rdx_24
150         var_221 = 0
151     }
152     else
153     {
154         int64_t rax_44
155         int64_t rdx_26
156         rax_44, rdx_26 = String.init(
157         _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
158         Invalid key ", 0xf, 0)
159         var_258 = rax_44
160         _swift_bridgeObjectRelease(rdx_21)
161         (&outlined destroy of String)(&var_48)
162         var_238 = var_258
163         var_230 = rdx_26
164         var_221 = 1
165     }
166     int64_t rcx_13
167     rcx_13.b = var_221
168     rcx_13.b = rcx_13.b & 1
169     return var_238

```

D.2 With ValueTypes Rules

```

1 int64_t KeyValidator.validate(_:)(int64_t arg1, int64_t
2 arg2, int64_t arg3, int64_t arg4) {
3     int512_t zmm0
4     zmm0.0x0 = 0x0
5     int128_t var_148 = 0x0
6     int256_t s_1
7     &__builtin_memset(&s_1, 0x0, 0x50)
8     int64_t var_f8 = 0x0
9     Type_3 rax_13
10    &__builtin_memset(&rax_13, 0x0, 0x30)
11    void* rax = type metadata accessor for SHA256Digest(0
12    x0, zmm0)
13    void* rax_1 = *(rax + 0xfffffffffffffffff8)
14    int64_t rax_2 = *(rax_1 + 0x40)
15    int64_t rdx_1
16    int64_t rsi_1
17    int64_t rdi_1
18    rdx_1, rsi_1, rdi_1 = &___chkstk_darwin()
19    Type_13 rax_44
20    void* rsp = &rax_44 - rax_2 + 0xf && 0
21    xfffffffffffffffff0
22    int128_t var_28
23    var_28.0x0 = rdi_1
24    var_28.0x8 = rsi_1
25    int128_t var_38
26    var_38.0x0 = rdx_1
27    var_38.0x8 = arg4
28    Type_1 rax_6 = &String.init(_builtinStringLiteral:
29    utf8CodeUnitCount:isASCII:)(&data_100007bd8, 0x0, 0
30    x1)
31    Type_1 rax_6
32    int64_t rax_7
33    int64_t rcx_1
34    int64_t rdx_3
35    int64_t r8_1

```

```

31     rax_7, rcx_1, rdx_3, r8_1 = String.makeIterator()(
32         arg1, arg2)
33     s_1.0x0 = rax_7
34     s_1.0x8 = rdx_3
35     s_1.0x10 = rcx_1
36     s_1.0x18 = r8_1
37     while (0x1) {
38         Type_2 rax_10 = String.Iterator.next()(&s_1)
39         if (rdx_4 == 0x0) {
40             break
41         }
42         Type_3 rax_13 = &DefaultStringInterpolation.init(
43             literalCapacity:interpolationCount:)(0x0, 0x2)
44         Type_4 rax_14 = &String.init(
45             _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
46             data_100007bd8, 0x0, 0x1)
47         &DefaultStringInterpolation.appendLiteral(_:)(&
48             rax_13, rax_14)
49         &DefaultStringInterpolation.appendInterpolation<A>
50         >(_:)(&rax_13, &rax_10, type metadata for Character,
51             protocol witness table for Character, protocol
52             witness table for Character)
53         Type_5 rax_16 = &String.init(
54             _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
55             data_100007bd8, 0x0, 0x1)
56         &DefaultStringInterpolation.appendLiteral(_:)(&
57             rax_13, rax_16)
58         &DefaultStringInterpolation.appendInterpolation<A>
59         >(_:)(&rax_13, &rax_17, type metadata for String,
60             protocol witness table for String, protocol witness
61             table for String)
62         Type_6 rax_19 = &String.init(
63             _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(&
64             data_100007bd8, 0x0, 0x1)
65         &DefaultStringInterpolation.appendLiteral(_:)(&
66             rax_13, rax_19)
67         Type_7 rax_21 = &String.init(stringInterpolation
68             :)(rax_13)
69         rax_6.0x0 = rax_21
70     }
71     int64_t rax_23 = type metadata accessor for SHA256(0
72         x0)
73     Type_8 rax_24 = String.utf8.getter(rax_6.0x0, rax_6.0
74         x8)
75     Type_9 rax_26 = Data.init<A>(_:)(&rax_24, type
76         metadata for String.UTF8View, &lazy protocol witness
77         table accessor for type String.UTF8View and
78         conformance String.UTF8View())
79     int64_t* var_1e0 = &rax_26
80     static HashFunction.hash<A>(data:)(&rax_26, rax_23,
81         type metadata for Data, &lazy protocol witness table
82         accessor for type SHA256 and conformance SHA256(),
83         &lazy protocol witness table accessor for type Data
84         and conformance Data())
85     &__chkstk_darwin()
86     *(rsp - 0x10) = &closure #1 in KeyValidator.validate(
87         _:)
88     *(rsp - 0x8) = 0x0
89     int64_t rax_33 = Sequence.compactMap<A>(_:)(&partial
90         apply for thunk for @callee_guaranteed (@unowned
91         UInt8) -> (@owned String?, @error @owned Error), rsp
92         - 0x20, rax, type metadata for String, &lazy
93         protocol witness table accessor for type
94         SHA256Digest and conformance SHA256Digest())
95     if (var_f8 != 0x0) {
96         trap(0x6)
97     }
98     *(rax_1 + 0x8)(rsp, rax, rax_33, rsp)
99     int64_t var_98 = rax_33

```

```

67     int64_t rax_36 = &
68     __swift_instantiateConcreteTypeFromMangledName(&
69     demangling cache variable for type metadata for [
70     String])
71     int64_t rax_37 = &lazy protocol witness table
72     accessor for type [String] and conformance [A]()
73     Type_10 rax_38 = &default argument 0 of
74     BidirectionalCollection<>.joined(separator:)(rax_36)
75     Type_11 rax_39 = BidirectionalCollection<>.joined(
76     separator:)(rax_38, rax_36, rax_37)
77     int128_t var_a8
78     var_a8.0x0 = rax_39
79     char rax_41 = static String.== infix(_:_:)(rax_39,
80     arg3, arg4)
81     Type_12 rax_43
82     char var_221
83     if (rax_41 && 0x1 != 0x0) {
84         Type_13 rax_44 = &String.init(
85             _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
86             "Access Granted ", 0x12, 0x0)
87         var_221 = 0x0
88     }
89     else {
90         Type_13 rax_44 = &String.init(
91             _builtinStringLiteral:utf8CodeUnitCount:isASCII:)(
92             "Invalid key ", 0xf, 0x0)
93         var_221 = 0x1
94     }
95     int64_t rcx_13
96     rcx_13.0x0 = var_221
97     rcx_13.0x0 = rcx_13.0x0 && 0x1
98     return rax_43
99 }

```

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.

```

1 func KeyValidator.validate(arg1: int64_t, arg2: int64_t,
2     arg3: int64_t, arg4: int64_t) -> int64_t {
3     int512_t zmm0
4     zmm0.0x0 = 0x0
5     int128_t var_148 = 0x0
6     int256_t s_1
7     &__builtin_memset(&s_1, 0x0, 0x50)
8     int64_t var_f8 = 0x0
9     Type_3 rax_13
10    &__builtin_memset(&rax_13, 0x0, 0x30)
11    void* rax = type metadata accessor for SHA256Digest(0
12        x0, zmm0)
13    void* rax_1 = *(rax + 0xfffffffffffffffff8)
14    int64_t rax_2 = *(rax_1 + 0x40)
15    int64_t rdx_1
16    int64_t rsi_1
17    int64_t rdi_1
18    rdx_1, rsi_1, rdi_1 = &__chkstk_darwin()
19    Type_13 rax_44
20    void* rsp = &rax_44 - rax_2 + 0xf && 0
21    xfffffffffffffffff0
22    int128_t var_28
23    var_28.0x0 = rdi_1
24    var_28.0x8 = rsi_1
25    int128_t var_38
26    var_38.0x0 = rdx_1
27    var_38.0x8 = arg4
28    var rax_6: String
29    for rax_10 in arg1 {
30        Type_1 rax_17 = rax_6

```



```

28     rax_6 = "\\(rax_10)\\(rax_17)"
29 }
30 int64_t rax_23 = type metadata accessor for SHA256(0
x0)
31 Type_8 rax_24 = String.utf8.getter(rax_6.0x0, rax_6.0
x8)
32 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())
33 int64_t* var_1e0 = &rax_26
34 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())
35 &__chkstk_darwin()
36 *(rsp - 0x10) = &closure #1 in KeyValidator.validate(
_:)
37 *(rsp - 0x8) = 0x0
38 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())
39 if (var_f8 != 0x0) {
40     trap(0x6)
41 }
42 *(rax_1 + 0x8)(rsp, rax, rax_33, rsp)
43 int64_t var_98 = rax_33

```

```

44 int64_t rax_36 = &
__swift_instantiateConcreteTypeFromMangledName(&
demangling cache variable for type metadata for [
String])
45 int64_t rax_37 = &lazy protocol witness table
accessor for type [String] and conformance [A]()
46 Type_10 rax_38 = &default argument 0 of
BidirectionalCollection<>.joined(separator:)(rax_36)
47 Type_11 rax_39 = BidirectionalCollection<>.joined(
separator:)(rax_38, rax_36, rax_37)
48 int128_t var_a8
49 var_a8.0x0 = rax_39
50 char rax_41 = static String.== infix(_:_:)(rax_39,
arg3, arg4)
51 Type_12 rax_43
52 char var_221
53 if (rax_41 && 0x1 != 0x0) {
54     Type_13 rax_44 = &String.init(
_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("
Access Granted ", 0x12, 0x0)
55     var_221 = 0x0
56 }
57 else {
58     Type_13 rax_44 = &String.init(
_builtinStringLiteral:utf8CodeUnitCount:isASCII:)("
Invalid key ", 0xf, 0x0)
59     var_221 = 0x1
60 }
61 int64_t rcx_13
62 rcx_13.0x0 = var_221
63 rcx_13.0x0 = rcx_13.0x0 && 0x1
64 return rax_43
65 }

```